МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

С. М. Порошин, А. М. Носик

ПРОГРАМУВАННЯ
Частина 1

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 004.4(75)
П59

Рецензенти:
М .А. Мірошник, д-р техн. наук, професор,
Харківський національний університет ім. В. Н. Каразіна;
Ю.Ф. Кучеренко, канд. техн. наук, старший науковий співробітник,
Харківський національний університет Повітряних Сил ім. 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 1: навчальний посібник / С. М. Порошин,
П59 А. М. Носик. – Харків: НТУ «ХПІ», 2024. – 377 с.

ISBN 978-617-05-0468-5

У навчальному посібнику викладено основні принципи з побудови обчислювальних алгоритмів, вивчення основ мови програмування високого рівня C 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 алгоритмізації обчислювальних процесів, досліджені базові типи мови C, робота з інструкціями вибору, масивами, стандартними функціями та функціями користувача, вказівниками, а також роботою з файлами.

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

УДК 004.4(75)

Іл. 132. Табл. 18. Бібліогр. 12 назв.

ISBN 978-617-05-0468-5

© Порошин С. М., Носик А. М., 2024
© НТУ «ХПІ», 2024

ЗМІСТ

ВСТУП 11
1. ОСНОВИ АЛГОРИТМІЗАЦІЇ 12
1.1. Алгоритми та мови програмува 12
1.1.1. Поняття алгоритму 12
1.1.2. Виконавець алгоритму 12
1.1.3. Властивості алгоритмів 13
1.1.4. Форми запису алгоритмів 14
1.2. Мови програмування 16
1.2.1. Рівень мови програмування 16
1.2.2. Переваги та недоліки машинних мов 17
1.2.3. Мова асемблера 17
1.2.4. Переваги алгоритмічних мов перед машинними 18
1.3. Розв’язання задач за допомогою комп’ютера 19
1.3.1. Основні етапи розв’язання завдань за допомогою комп’ютера 19
1.3.2. Математична модель 20
1.3.3. Основні етапи процесу розробки програм 21
1.3.4. Контроль тексту програми до виходу на комп’ютер 22
1.3.5. Відлагодження програм 23
1.3.6. Тестування програми 24
1.3.7. Помилки в програмах 26
1.4. Схеми алгоритмів 29
1.4.1. Види алгоритмів 29
1.4.2. Структурне зображення алгоритму 30
1.4.3. Правила виконання з’єднань 38

1.4.4. Приклади використання схем алгоритмів 39
Контрольні запитання та завдання 43
Завдання для самостійного розв’язання 44
2. ОСНОВИ ПРОГРАМУВАННЯ 45
2.1. Вступ до програмування 45
2.2. Сфери застосування програмування 46
2.2.1. Веб-розробка 46
2.2.2. Мобільна розробка.. 47
2.2.3. Десктопні додатки 47
2.2.4. Розробка ігор 48
2.2.5 Data Science 48
2.2.6.Програмування вбудованих систем 49
2.2.7. Інтернет речей (Internet of Things, IoT) 49
2.2.8. Хмарні обчислення 50
2.3. Парадигми програмування 52
2.3.1. Імперативне програмування 54
2.3.2. Декларативне програмування 55
2.3.3. Процедурне (алгоритмічне) програмування................................ 55
2.3.4. Структурне програмування.. 57
2.3.5. Аспектно-орієнтоване програмування... 58
2.3.6. Об’єктно-орієнтоване програмування (ООП) 59
2.3.7. Функціональне програмування.. 59
2.3.8. Логічне програмування 60
2.3.9. Узагальнене програмування 60
2.4. Інструменти для ефективної роботи програміста 61

2.4.1. Інтегроване середовище розробки 61
2.4.2. Профілювальник коду (профайлер) 62
2.4.3. Система контролю версій... 62
2.4.4. Візуальний редактор інтерфейсу 63
2.4.5. Редактор баз даних 63
2.4.6. Інструмент тестування ПЗ.. 64
2.4.7. Фреймворки.. 65
Контрольні запитання та завдання 65
3. ОСНОВИ ПРОГРАМУВАННЯ НА МОВІ *С* 67
3.1. Загальне знайомство з мовою *С*... 67
3.3.1. Походження мови *С*.. 67
3.1.2. Стандарти мови С 67
3.1.3. Загальна інформація про С... 69
3.1.4. Переваги мови С.. 71
3.1.5. Застосування мови *С*, які мають високі вимоги до продуктивності ... 72
3.1.6. Майбутнє мови *С* 73
3.1.7. Базові поняття мови *С* 73
3.2. Базові поняття мови *С* 79
3.2.1. Константи.. 79
3.2.2. Рядки (рядкові константи) 85
3.2.3. Змінні та іменовані константи 87
3.3. Операції та вирази мови С.. 98
3.3.1. Знаки операцій.. 98
3.3.3. Вирази та приведення арифметичних типів... 114
3.3.4. Відношення та логічні вирази... 116

3.3.5. Присвоювання (вираз і оператор) 117
3.3.6. Приведення типів 119
3.3.7. Вирази з порозрядними операціями 122
3.3.8. Умовний вираз 125
3.4. Функції printf() і scanf() 126
3.4.1. Використання функцій printf() і scanf() 126
3.4.2. Функція printf() 126
3.4.3. Функція scanf() 139
3.4.4. Модифікатор * у функціях printf() і scanf() 145
Контрольні запитання та завдання 147
Завдання для самостійного розв’язання 148
4. КЕРУЮЧІ СТРУКТУРИ 150
4.1. Інструкції вибору в *C* 150
4.1.1. Інструкція вибору if 151
4.1.2. Інструкція вибору if...else 153
4.1.3. Умовний оператор та умовний вираз 153
4.1.4. Вкладені інструкції if...else 155
4.2. Інструкція множинного вибору switch 158
4.3. Різновиди циклів в мові C 161
4.3.1. Цикл з передумовою while 162
4.3.2. Цикл з постумовою do...while 164
4.3.3. Цикл з лічильником for 165
4.3.4. Цикл з виходом з середини 169
4.3.5. Нескінченний цикл 171
4.4. Пропуск ітерації continue 174

4.5. Необхідність використання дострокового виходу з циклу та пропуску ітерації 175
4.6. Вкладені цикли 176
Контрольні запитання та завдання 178
Завдання для самостійного розв’язання 179
5. ОРГАНІЗАЦІЯ ОДНОТИПНИХ ДАНИХ 180
5.1. Масиви 180
5.1.1. Масиви в мові *C* 180
5.1.2. Використання циклів for з масивами 183
5.1.3. Ініціалізація масивів 185
5.1.4. Призначені ініціалізатори (C99) 191
5.1.5. Присвоювання значень елементам масиву 193
5.1.6. Межі масиву 194
5.1.7. Зазначення розміру масиву 196
5.1.8. Двовимірні масиви 198
5.1.9. Ініціалізація двовимірного масиву 201
5.1.10. Масиви з великою кількістю вимірювань 202
Контрольні запитання та завдання 203
Завдання для самостійного розв’язання 204
6. ФУНКЦІЇ ТА ВКАЗІВНИКИ 205
6.1. Функції 205
6.1.1. Поняття функції 205
6.1.2. Створення та використання простої функції 207
6.1.3. Аргументи функції 210
6.1.4. Визначення функції з аргументами: формальні параметри 212

6.1.5. Створення прототипу функції з аргументами ... 213
6.1.6. Виклик функції з аргументами: фактичні аргументи ... 213
6.1.7. Представлення функції у вигляді чорного ящика ... 214
6.1.8. Повернення значення з функції за допомогою return ... 215
6.1.9. Типи функцій ... 216
6.1.10. Створення прототипів функцій в ANSIC ... 218
6.1.11. Рішення стандарту ANSI C ... 218
6.1.12. Відсутність аргументів і невизначені аргументи ... 220
6.1.13. Перевага прототипів ... 221
6.1.14. Рекурсія ... 222
6.2. Вказівники ... 233
6.2.1. Змінні-вказівники, визначення та ініціалізація ... 234
6.2.2. Оператори вказівників ... 235
6.2.3. Передача аргументів функціям за посиланням ... 237
6.2.4. Використання кваліфікатора constз вказівниками ... 241
6.2.5. Вирази з вказівниками та арифметика вказівників ... 255
6.2.6. Зв'язок між вказівниками та масивами ... 259
6.2.7. Вказівники на функції ... 266
Контрольні запитання та завдання ... 273
Завдання для самостійного розв’язання ... 274
7. ФУНКЦІЇ ДЛЯ РОБОТИ З РЯДКАМИ ... 276
7.1. Рядки. Функції для вводу-виводу рядків ... 276
7.1.1. Рядки та ввід-вивід рядків ... 276
7.1.2. Визначення рядків у програмі ... 277
7.1.3. Масиви символьних рядків ... 284

7.1.4. Вказівники та рядки 286
7.1.5. Функції для вводу рядків 287
7.1.6. Функції для виводу рядків 303
7.1.7. Можливість самостійного створення функцій вводу-виводу 307
7.2. Функції для роботи з рядками 310
7.2.1. Функція strlen() 310
7.2.2. Функція strcat() 312
7.2.3. Функція strncat( ) 313
7.2.4. Функція strcmp() 316
7.2.5. Функції strcpy()і strncpy() 320
7.2.6. Функція sprintf() 326
7.2.7. Інші функції для роботи з рядками 328
7.2.8. Сортування рядків 331
Контрольні запитання та завдання 335
Завдання для самостійного розв’язання 337
8. ФАЙЛОВИЙ ВВІД-ВИВІД 338
8.1. Функції для роботи з файлами 338
8.1.1. Взаємодія з файлами 338
8.1.2. Стандартний ввід-вивід 340
8.1.3. Файловий ввід-вивід: fprintf(), fscanf(), fgets()і fputs() 348
8.1.4. Інші стандартні функції вводу-виводу 359
8.2. Двійковий файловий ввід-вивід 361
8.2.1. Двійковий ввід-вивід: функції fread()і fwrite() 361
8.2.2. Функція fwrite(). 363
8.2.3. Функція fread() 364

8.2.4. Функції feof() і ferror() 365
8.2.5. Довільний доступ з двійковим вводом-виводом 371
8.2.6. Ключові поняття 373
Контрольні запитання та завдання 374
Завдання для самостійного розв’язання 375
ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ 376

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 *C* та розробки комп’ютерних програм, ознайомлення з методами розв’язання обчислювальних задач та формування вмінь з їх практичного використання.

Багато фірм, що розробляють програмне забезпечення, все частіше звертаються до *C* як до зручної мови для реалізації своїх проєктів, оскільки відомо, що *C* дозволяє отримати компактні та ефективні програми. Мобільність і продуктивність мови *C* є одними з найбільш затребуваних її характеристик при реалізації таких операційних систем як Linux, Microsoft Windows, а також Google Android. Мова *C* 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.

Успіх у вивченні тієї чи іншої мови програмування неможливий без закріплення отриманих знань. Представлені матеріали навчального посібника напралені на опрацювання питань, пов’язаних з роботою в інтегрованому середовищі розробки *Code::Blocks*, розглядом основних принципів алгоритмізації обчислювальних процесів, дослідженням базових типів мови *C*, роботою з інструкціями вибору, масивами, стандартними функціями та функціями користувача, вказівниками, а також роботою з файлами та призначені для опанування дисципліни «Програмування».

1. ОСНОВИ АЛГОРИТМІЗАЦІЇ

1.1. Алгоритми та мови програмування

1.1.1. Поняття алгоритму

В математиці для вирішення типових задач використовують певні правила, що описують послідовність деяких дій. Наприклад, правила складання дробових чисел, рішення квадратних рівнянь та інше. Зазвичай будь-які інструкції та правила являють собою послідовність дій, які необхідно виконати в певному порядку. Для вирішення завдання треба знати, що дано, що слід отримати і які дії, і в якому порядку слід для цього виконати. Припис, що визначає порядок виконання дій над даними з метою отримання необхідних результатів, і є алгоритмом.

Алгоритм (лат. `Algorithmi` – від арабського імені математика Аль-Хорезмі) – це кінцева сукупність точно заданих правил рішення довільного класу задач або набір інструкцій, що описують порядок дій виконавця для вирішення деякої задачі.

Незалежні інструкції можуть виконуватися в довільному порядку або паралельно, якщо це дозволяють виконавці алгоритму.

1.1.2. Виконавець алгоритму

Виконавець алгоритму – це деяка абстрактна чи реальна (технічна, біологічна або біотехнічна) система, яка здатна виконати дії, що передбачені алгоритмом.

Виконавця характеризують:

- середовище;
- елементарні дії;
- система команд;
- відмови.

Середовище (або обстановка) – це «місце проживання» виконавця. **Система команд**. Кожен виконавець може виконувати команди тільки з

деякого суворо заданого списку – системи команд виконавця. Для кожної команди повинні бути задані умови застосування (в яких станах середовища може бути виконана команда) і описані результати виконання команди.

Після виклику команди виконавець здійснює відповідну **елементарну дію**.

Відмови виконавця виникають, якщо команда викликається при неприпустимому для неї стані середовища.

Зазвичай виконавець нічого не знає про мету алгоритму. Він виконує всі отримані команди, не ставлячи питань «Чому?» і «Навіщо?». **У програмуванні універсальним виконавцем алгоритмів є комп'ютер.**

1.1.3. Властивості алгоритмів

- зрозумілість для виконавця; - дискретність;
- визначеність;
- результативність;
- масовість.

Зрозумілість для виконавця – виконавець алгоритму повинен розуміти, як його виконувати. Іншими словами, маючи алгоритм і довільний варіант початкових даних, виконавець повинен знати, як треба діяти для виконання цього алгоритму.

Дискретність (переривчастість, роздільність) – алгоритм повинен представляти процес вирішення завдання як послідовне виконання простих (або раніше визначених) кроків (етапів).

Визначеність – кожне правило алгоритму має бути чітким, однозначним і не залишати місця для іншого трактування. Завдяки цій властивості виконання алгоритму носить механічний характер і не вимагає ніяких додаткових вказівóк або відомостей про розв'язувану задачу.

Результативність (або скінченність) полягає в тому, що за кінцеве число кроків алгоритм або повинен призводити до вирішення завдання, або після кінцевого числа кроків зупинятися через неможливість отримати рішення з

видачею відповідного повідомлення, або необмежено тривати протягом часу, відведеного для виконання алгоритму, з видачею проміжних результатів.

Масовість означає, що алгоритм вирішення задачі розробляється в загальному вигляді, тобто він повинен бути застосовний для деякого класу задач, що розрізняються лише початковими даними. При цьому початкові дані можуть вибиратися з деякої області, яка називається *областю застосовності алгоритму*.

1.1.4. Форми запису алгоритмів

На практиці найбільш поширеними є такі форми подання алгоритмів:

- **словесна** (запис на природній мові);
- **графічна** (зображення з графічних символів);
- **псевдокоди** (напівформалізовані описи алгоритмів на умовній алгоритмічній мові, що включають в себе як елементи мови програмування, так і фрази природної мови, загальноприйняті математичні позначення та ін.);
- **програмна** (тексти на мовах програмування).

Словесний спосіб записи алгоритмів – це опис послідовних етапів обробки даних. Алгоритм задається у довільному вигляді природною мовою. Словесний спосіб є не досить поширеним, бо він:

- строго не формалізуються;
- страждає багатослівністю записів;
- припускає неоднозначність тлумачення окремих приписів (рос. предписаний).

Графічний спосіб зображення алгоритмів є більш компактним і наочним порівняно зі словесним.

При графічному зображенні алгоритм подається у вигляді послідовності пов’язаних між собою функціональних блоків, кожен з яких відповідає виконанню однієї або декількох дій.

Таке графічне зображення називається **схемою алгоритму** або **блок-схемою.** У блок-схемі кожному типу дій (введення початкових даних,

обчислення значень виразів, перевірка умов, керування повторенням дій, закінчення обробки та ін.) відповідає геометрична фігура, яка представлена у вигляді блочного символу. Блокові символи з'єднуються лініями переходів, що визначають черговість виконання дій.

Псевдокод являє собою систему позначень і правил, яка призначена для однакового запису алгоритмів.

Псевдокод займає проміжне місце між природною і формальною мовами. З одного боку, він близький до звичайної природної мови, тому алгоритми можуть на ньому записуватися й читатися як звичайний текст. З іншого боку, в псевдокоді використовуються деякі формальні конструкції і математична символіка, що наближає запис алгоритму до загальноприйнятого математичного запису.

У псевдокоді не прийнято використовувати строгі синтаксичні правила для запису команд, які властиві формальним мовам. Це полегшує запис алгоритму на стадії його проектування та надає можливість використовувати більш широкий набір команд, розрахований на абстрактного виконавця.

Однак в псевдокоді зазвичай є деякі конструкції, які властиві формальним мовам. Це полегшує перехід від запису на псевдокоді до запису алгоритму на формальній мові. Зокрема, в псевдокоді, так само, як і в формальних мовах, є службові слова, зміст яких визначено раз і назавжди. Вони виділяються в друкованому тексті жирним шрифтом, а в рукописному тексті підкреслюються.

Єдиного або формального визначення псевдокоду не існує, тому можливі різні псевдокоди, що відрізняються набором службових слів і основних (базових) конструкцій.

На практиці в якості виконавців алгоритмів використовуються спеціальні автомати – комп'ютери, тому алгоритм, що призначений для виконання на комп'ютері, повинен бути записаний на зрозумілій йому мові. Тут на перший план постає необхідність точного запису команд, яка не залишає місця для довільного тлумачення їх виконавцем.

Мова для запису алгоритмів повинна бути формалізованою.

Таку мову прийнято називати **мовою програмування**, а запис алгоритму на цій мові – **програмою** для комп'ютера.

Алгоритмічну мову утворюють: *алфавіт*, *синтаксис* і *семантика*. **Алфавіт** – це фіксований для даної мови набір основних символів, з яких повинен складатися будь-який текст програми.

Синтаксис – це набір правил, що встановлюють допустимі комбінації символів на цій мові.

Семантика – сукупність правил, що визначають зміст синтаксично коректних конструкцій мови, її зміст.

1.2. Мови програмування

1.2.1. Рівень мови програмування

На даний час у світі існує кілька сотень реально використовуваних мов програмування (МП). Для кожної є своя сфера застосування.

Будь-який алгоритм, як ми знаємо, є послідовністю приписів, виконавши які можна за кінцеве число кроків перейти від початкових даних до результату.

Залежно від ступеня деталізації приписів визначається рівень мови програмування – чим менше деталізація, тим вище рівень.

За цим критерієм можна виділити такі рівні МП: - машинні;

- машинно-орієнтовані (асемблери);

- машинно-незалежні (мови високого рівня).

Машинні мови та машинно-орієнтовані мови – це мови низького рівня, що вимагають зазначення дрібних деталей процесу обробки даних.

Мови програмування високого рівня імітують природні мови, використовуючи деякі слова розмовної мови та загальноприйняті математичні символи. Ці мови є більш зручними для людини.

Мови високого рівня поділяються на:

– процедурні (`Basic`, `Pascal`, `C` та ін.);

– логічні (Prolog, Lispi ін.);

– об'єктно-орієнтовані (C++,C#, Java та ін.).

Процедурні (алгоритмічні) мови призначені для однозначного опису алгоритмів.

Для вирішення завдання процедурні мови вимагають в тій чи іншій формі явно записати процедуру її рішення.

Логічні (декларативні) мови орієнтовані на систематичний і формалізований опис задачі.

В основі об’єктно-орієнтованих мов існує поняття об’єкта, що поєднує в собі дані та дії над ними.

Програма на **об’єктно-орієнтованій мові**, вирішуючи деяку задачу, по суті, описує частину світу, що відноситься до цього завдання. Опис дійсності в формі системи взаємодіючих об’єктів природніше, ніж у формі взаємодіючих процедур.

1.2.2. Переваги та недоліки машинних мов

Кожен комп’ютер має свою машинну мову, тобто свою сукупність машинних команд, яка відрізняється кількістю адрес в команді, призначенням інформації, що задається в адресах, набором операцій, які може виконати машина та ін.

Переваги: під час програмування на машинній мові програміст може тримати під своїм контролем кожну команду і кожну комірку пам’яті, використовувати всі можливості наявних машинних операцій.

Недоліки: процес написання програми на машинній мові дуже трудомісткий і виснажливий, програма виходить громіздкою, важкою для перегляду, її важко відлагоджувати, змінювати та розвивати.

Тому в разі, коли потрібно мати ефективну програму, яка повинна максимально враховувати специфіку конкретного комп’ютера, замість машинних мов використовують близькі до них машинно-орієнтовані мови (**асемблери**).

1.2.3. Мова асемблера

Мова асемблера – це машинно-залежна мова низького рівня, в якій

короткі мнемонічні імена відповідають окремим машинним командам.

Використовується для подання в зрозумілій формі програм, записаних в машинному коді.

Мова асемблера дозволяє програмісту користуватися текстовими мнемонічними (тобто такими, що легко запам’ятовуються людиною) кодами, на свій розсуд присвоювати символічні імена регістрів комп’ютера і пам’яті, а також задавати зручні для себе способи адресації. Крім того, вона дозволяє використовувати різні системи числення (наприклад, десяткову або шістнадцяткову) для представлення числових констант, використовувати в програмі коментарі та ін.

Програми, які написані на мові асемблера, вимагають значно меншого об’єму пам’яті та часу виконання. Знання програмістом мови асемблера та машинного коду дає йому розуміння архітектури машини.

Незважаючи на те, що більшість фахівців в сфері програмного забезпечення розробляють програми на мовах високого рівня, найбільш потужне й ефективне програмне забезпечення повністю або частково написано на мові асемблера.

Мови високого рівня були розроблені для того, щоб звільнити програміста від урахування технічних особливостей конкретних комп’ютерів, їх архітектури. На противагу цьому, мова асемблера розроблена з метою врахування конкретної специфіки процесора. Отже, для того, щоб написати програму на мові асемблера для конкретного комп’ютера, важливо знати його архітектуру.

Переведення програми з мови асемблера на машинну мову здійснюється спеціальною програмою, яка називається **асемблером** і є, по суті, найпростішим **транслятором**.

1.2.4. Переваги алгоритмічних мов перед машинними

Основні переваги такі:

- *алфавіт алгоритмічної мови є значно ширшим за алфавіт машинної*

мови, що істотно підвищує наочність тексту програми;

- *набір операцій, припустимих для використання, не залежить від набору машинних операцій*, а вибирається з міркувань зручності формулювання алгоритмів розв'язання задач певного класу;

- *формат речень є досить гнучким і зручним для використання*, що дозволяє за допомогою одного речення задати досить змістовний етап обробки даних;

- *необхідні операції задаються за допомогою загальноприйнятих математичних позначень*;

- *даними в алгоритмічних мовах присвоюються індивідуальні імена*, які обирає сам програміст;

- *в алгоритмічній мові може бути передбачений значно ширший набір типів даних* у порівнянні з набором машинних типів даних.

Таким чином, алгоритмічні мови значною мірою є машинно-незалежними. Вони полегшують роботу програміста та підвищують надійність створюваних програм.

1.3. Розв'язання задач за допомогою комп'ютера

1.3.1. Основні етапи розв'язання завдань за допомогою комп'ютера

Рішення задач за допомогою комп'ютера включає в себе наступні основні етапи, частина з яких здійснюється без участі комп'ютера.

1) Постановка завдання:

- збір інформації про завдання; - формулювання умови задачі;

- визначення кінцевих цілей рішення задачі; - визначення форми видачі результатів;

- опис даних (їх типів, діапазонів величин, структури та ін.).

2) Аналіз і дослідження задачі, моделі: - аналіз існуючих аналогів;

- аналіз технічних і програмних засобів; - розробка математичної моделі;

- розробка структур даних. **3) Розробка алгоритму:**

- вибір методу проектування алгоритму;

- вибір форми запису алгоритму (блок-схема, псевдокод та ін.); - вибір тестів і методу тестування;

- проектування алгоритму. **4) Програмування:**

- вибір мови програмування;

- уточнення способів організації даних;

- запис алгоритму обраною мовою програмування. **5) Тестування та відлагодження:**

- синтаксичне відлагодження;

- відлагодження семантики та логічної структури;

- тестові розрахунки та аналіз результатів тестування; - вдосконалення програми.

Аналіз результатів рішення задачі й уточнення в разі потреби математичної моделі з повторним виконанням етапів 2 – 5.

6) Супровід програми:

- доопрацювання програми для рішення конкретних задач;

- складання документації до вирішуваної задачі, математичної моделі, алгоритму, програми, набору тестів, використання.

1.3.2. Математична модель

Математична модель – це система математичних співвідношень – формул, рівнянь, нерівностей та інших, що відображають істотні властивості об’єкта чи явища.

Щоб описати явище, необхідно виявити найістотніші його властивості, закономірності, внутрішні зв’язки, роль окремих характеристик явища. Виділивши найбільш важливі фактори, можна знехтувати менш суттєвими.

Найбільш ефективно математичну модель можна реалізувати на комп’ютері у вигляді алгоритмічної моделі – так званого «обчислювального експерименту».

Звичайно, результати обчислювального експерименту можуть виявитися

й такими, що не відповідають дійсності, якщо в моделі не будуть враховані якісь важливі аспекти.

Створюючи математичну модель для вирішення задачі, потрібно:

1) виділити припущення, на яких буде ґрунтуватися математична модель;
2) визначити, що вважати початковими даними та результатами;

3) записати математичні співвідношення, що зв'язують результати з початковими даними.

При побудові математичних моделей далеко не завжди вдається знайти формули, що явно виражають шукані величини через дані. У таких випадках використовуються математичні методи, що дозволяють дати відповіді того чи іншого ступеня точності.

Існує не тільки **математичне моделювання** будь-якого явища, а й **візуально-натурне моделювання**, яке забезпечується за рахунок відображення цих явищ засобами машинної графіки, тобто перед дослідником демонструється своєрідний «комп'ютерний мультфільм», що знімається в реальному масштабі часу. Наочність тут дуже висока.

1.3.3. Основні етапи процесу розробки програм

На початковому етапі роботи аналізуються та формулюються вимоги до програми, розробляється точний опис того, що повинна робити програма і яких результатів необхідно досягти за її допомогою.

Потім програма розробляється з використанням тієї чи іншої технології програмування (наприклад, структурного програмування).

Отриманий варіант програми піддається систематичному тестуванню – адже наявність помилок у щойно розробленій програмі це цілком нормальне закономірне явище.

Практично неможливо скласти реальну (досить складну) програму без помилок. Не можна робити висновок, що програма правильна, лише на тій підставі, що вона не відкинута машиною і видала результати. Все, що досягнуто в цьому випадку, це отримання якихось результатів, не обов'язково

правильних. У програмі при цьому може залишатися велика кількість логічних помилок. Відповідальні ділянки програми перевіряються з використанням методів доведення правильності програм.

Для кожної програми обов'язково проводяться роботи по забезпеченню якості й ефективності програмного забезпечення, аналізуються та поліпшуються часові характеристики.

1.3.4. Контроль тексту програми до виходу на комп’ютер

Текст програми можна проконтролювати за столом за допомогою перегляду, перевірки та прокрутки.

1) **Перегляд**. Текст програми переглядається на предмет виявлення описок і розбіжностей з алгоритмом.

Потрібно переглянути організацію всіх циклів, щоб переконатися в правильності операторів, які задають кратності циклів. Корисно подивитися ще раз умови в умовних операторах, аргументи в зверненнях до підпрограм та ін.

2) **Перевірка**. Під час перевірки програми програміст за текстом подумки намагається відновити той обчислювальний процес, який вона визначає, після чого звіряє його з необхідним процесом.

На час перевірки потрібно «забути», що повинна робити програма, і «дізнаватися» про це по ходу її перевірки. Тільки після закінчення перевірки програми можна «згадати» про те, що вона повинна робити та порівняти реальні дії програми з необхідними.

3) **Прокрутка**. Основою прокрутки є імітація програмістом за столом виконання програми на комп’ютері.

Для виконання прокрутки доводиться задаватися деякими початковими даними й виконувати між ними необхідні обчислення.

Прокрутка – це трудомісткий процес, тому її слід застосовувати лише для контролю логічно складних ділянок програм. Початкові дані повинні вибиратися такими, щоб до прокрутки залучалося більшість гілок програми.

1.3.5. Відлагодження програм

Відлагодження програми – це процес виявлення й усунення несправностей в програмі, вироблений за результатами її прогону на комп'ютері.

Тестування (англ. `test` – випробування) – це випробування, перевірка правильності роботи програми в цілому або її складових частин.

Відлагодження та тестування – це два чітко помітних і несхожих один на одного етапи:

- при відлагодженні відбувається локалізація й усунення синтаксичних помилок і явних помилок кодування;

- в процесі ж тестування перевіряється працездатність програми, яка не містить явних помилок.

Тестування встановлює факт наявності помилок, а відлагодження з'ясовує їх причину.

Англійський термін **`debugging`** («відлагодження») буквально означає «виловлювання жучків». Термін з'явився в 1945 р., коли один з перших комп'ютерів – «Марк-1» припинив роботу через те, що в його електричні кола потрапив метелик і заблокував своїми залишками одне з тисяч реле машини.

В сучасних програмних системах відлагодження здійснюється часто з використанням спеціальних програмних засобів, які називаються відлагоджувачами та дозволяють досліджувати внутрішню поведінку програми.

Програма-відлагоджувач зазвичай забезпечує наступні можливості:

- покрокове виконання програми з зупинкою після кожної команди (оператора);

- перегляд поточного значення будь-якої змінної або знаходження значення будь-якого виразу, в тому числі, з використанням стандартних функцій (при необхідності можна встановити нове значення змінної);

- встановлення в програмі «контрольних точок», тобто точок, в яких програма тимчасово припиняє своє виконання для оцінювання проміжних результатів та ін.

При відлагодженні програм важливо пам’ятати наступне:

- на початку процесу відлагодження треба використовувати прості тестові дані;

- труднощі, що виникають, слід чітко розділяти й усувати строго по черзі;

- не потрібно вважати причиною помилок комп’ютер, оскільки сучасні комп’ютери та транслятори мають надзвичайно високу надійність.

1.3.6. Тестування програми

Як би не була ретельно відлагоджена програма, вирішальним етапом, що встановлює її придатність для роботи, є контроль програми за результатами її виконання на системі тестів.

Програму умовно можна вважати правильною, якщо її запуск для обраної системи тестових даних у всіх випадках дає правильні результати.

Але, як справедливо вказував відомий теоретик програмування Е. Дейкстра, тестування може показати лише наявність помилок, але не їх відсутність. Нерідкими є випадки, коли нові вхідні дані викликають «відмову» або отримання невірних результатів роботи програми, яка вважалася повністю відлагодженою.

Для реалізації методу тестів повинні бути виготовлені або заздалегідь відомі еталонні результати. Обчислювати еталонні результати потрібно обов’язково до, а не після отримання машинних результатів. В іншому випадку є небезпека підгонки обчислюваних значень під бажані результати, які були отримані на комп’ютері раніше.

Тестові дані повинні забезпечити перевірку всіх можливих умов виникнення помилок:

- повинна бути випробувана кожна гілка алгоритму;

- черговий тестовий прогін повинен контролювати щось таке, що ще не було перевірено на попередніх прогонах;

- перший тест повинен бути максимально простим, щоб перевірити, чи працює програма взагалі;

- арифметичні операції в тестах повинні гранично спрощуватися для зменшення обсягу обчислень;

- кількість елементів послідовностей, точність для ітераційних обчислень, кількість проходів циклу в тестових прикладах повинні задаватися з міркувань скорочення обсягу обчислень;

- мінімізація обчислень не повинна знижувати надійності контролю;

- тестування повинно бути цілеспрямованим і систематизованим, оскільки випадковий вибір початкових даних призвів би до труднощів у визначенні ручним способом очікуваних результатів. Крім того, при випадковому виборі тестових даних можуть виявитися неперевіреними багато ситуацій;

- ускладнення тестових даних має відбуватися поступово.

Процес тестування можна розділити на три етапи:

1) Перевірка в нормальних умовах.

Передбачає тестування на основі даних, які характерні для реальних умов функціонування програми.

2) Перевірка в екстремальних умовах.

Тестові дані включають граничні значення області зміни вхідних змінних, які повинні сприйматися програмою як правильні дані. Типовими прикладами таких значень є дуже маленькі або дуже великі числа або відсутність даних. Ще один тип екстремальних умов – це граничні об’єми даних, коли масиви складаються з дуже малого або занадто великого числа елементів.

3) Перевірка у виняткових ситуаціях.

Проводиться з використанням даних, значення яких лежать за межами допустимої області змін. Відомо, що всі програми розробляються з розрахунком на обробку якогось обмеженого набору даних. Тому важливо отримати відповідь на наступні питання:

- що станеться, якщо програмі, яка не розрахована на обробку від’ємних і нульових значень змінних, в результаті якої-небудь помилки доведеться мати справу саме з такими даними?

- як буде вести себе програма, що працює з масивами, якщо кількість їх

елементів перевищить величину, що зазначена в оголошенні масиву?

- що станеться, якщо числа будуть занадто малими або занадто великими? Найгірша ситуація складається тоді, коли програма сприймає невірні дані

як правильні та видає невірний, але правдоподібний результат.

Програма повинна сама відкидати будь-які дані, які вона не в змозі обробляти правильно.

1.3.7. Помилки в програмах

Помилки можуть бути допущені на всіх етапах виконання завдання – від його постановки до оформлення. Різновиди помилок і відповідні приклади наведені в табл. 1.1.

Таблиця 1.1 – Види помилок

Вид помилок	Приклад
Неправильна постановка задачі	Правильне рішення невірно сформульованої задачі
Невірний алгоритм	Вибір алгоритму, що приводить до неточного або неефективного вирішення завдання
Помилка аналізу	Неповне урахування ситуацій, які можуть виникнути; логічні помилки
Семантичні помилки	Нерозуміння порядку виконання операторів
Синтаксичні помилки	Порушення правил, що визначаються мовою програмування
Помилки при виконанні операцій	Занадто велике число; ділення на нуль; добування квадратного кореня з від'ємного числа та ін.
Помилки в даних	Невдале визначення можливого діапазону зміни даних
Друкарська помилка	Переплутані близькі за написанням символи, наприклад, цифра **1**(одиниця) і букви **I** або **l** (л)
Помилки вводу-виводу	Неправильне зчитування вхідних даних; невірне задавання форматів даних

Зазвичай *синтаксичні помилки* виявляються на етапі трансляції. Багато ж інших помилок транслятора виявити неможливо, в наслідок того, що

транслятору невідомі задуми програміста.

Відсутність повідомлень машини про синтаксичні помилки є необхідною, але не достатньою умовою, щоб вважати програму правильною.

Приклади синтаксичних помилок:

- пропуск знака пунктуації;
- неузгодженість дужок;
- неправильне формування оператора;
- невірне утворення імен змінних;
- невірне написання службових слів;
- відсутність умов закінчення циклу;
- відсутність опису масиву та інші.

Існує безліч помилок, які транслятор виявити не в змозі, якщо оператори програми сформовані правильно.

Приклади таких помилок.

1) Логічні помилки:

- неправильне зазначення гілки алгоритму після перевірки деякої умови;
- неповне урахування можливих умов;
- пропуск в програмі одного або більшої кількості блоків алгоритму.

2) Помилки в циклах:

- неправильне зазначення початку циклу;
- неправильне зазначення умов закінчення циклу;
- неправильне зазначення числа повторень циклу;
- нескінчений цикл.

3) Помилки вводу-виводу; помилки при роботі з даними:

- неправильне задавання типу даних;
- організація зчитування меншого або більшого об’єму даних, ніж потрібно;
- неправильне редагування даних.

4) Помилки у використанні змінних:

- використання змінних без вказівки їх початкових значень;

- помилкове зазначення однієї змінної замість іншого.

5) Помилки при роботі з масивами:

- масиви попередньо не були обнулені;

- масиви неправильно описані;

- індекси йдуть у неправильному порядку.

6) Помилки в арифметичних операціях:

- неправильне зазначення типу змінної (наприклад, цілочислового типу – замість дійсного типу);

- неправильне визначення порядку дій;

- ділення на нуль;

- добування квадратного кореня з від’ємного числа;

- втрата значущих розрядів числа.

Всі ці помилки виявляються за допомогою тестування.

Супровід програм – це роботи, що пов’язані з обслуговуванням програм в процесі їх експлуатації.

Багаторазове використання розробленої програми для розв’язання різних завдань заданого класу вимагає проведення наступних додаткових робіт:

- виправлення виявлених помилок;

- модифікація програми для врахування нових експлуатаційних вимог; - доробка програми для вирішення конкретних завдань;

- проведення додаткових тестових розрахунків; - внесення виправлень до робочої документації; - удосконалення програми та інші.

Відносно від багатьох інших роботи із супроводу програм поглинають більш ніж половину витрат, що припадають на весь період часу існування програми (починаючи від вироблення початкової концепції й закінчуючи моральним її старінням) у вартісному вираженні.

Програма, що призначена для тривалої експлуатації, повинна мати відповідну документацію та інструкцію щодо її використання.

1.4. Схеми алгоритмів

1.4.1. Види алгоритмів

Будь-який обчислювальний процес може бути представлений комбінацією елементарних алгоритмічних структур, які можна розділити на три основних види:

- лінійні;
- розгалужені;
- циклічні.

Лінійний алгоритм – це набір команд (вказівок), що виконуються послідовно в часі один за одним (рис. 1.1, а).

Розгалужений алгоритм – це алгоритм, який містить хоча б одну умову, в результаті перевірки якої може здійснюватися поділ на кілька альтернативних гілок алгоритму (рис. 2.1, б).

Циклічний алгоритм – це алгоритм, який передбачає багаторазове повторення будь-якої дії (операцій) над новими вихідними даними (рис. 1.1, в). До циклічних алгоритмів зводиться більшість методів обчислень, перебору варіантів.

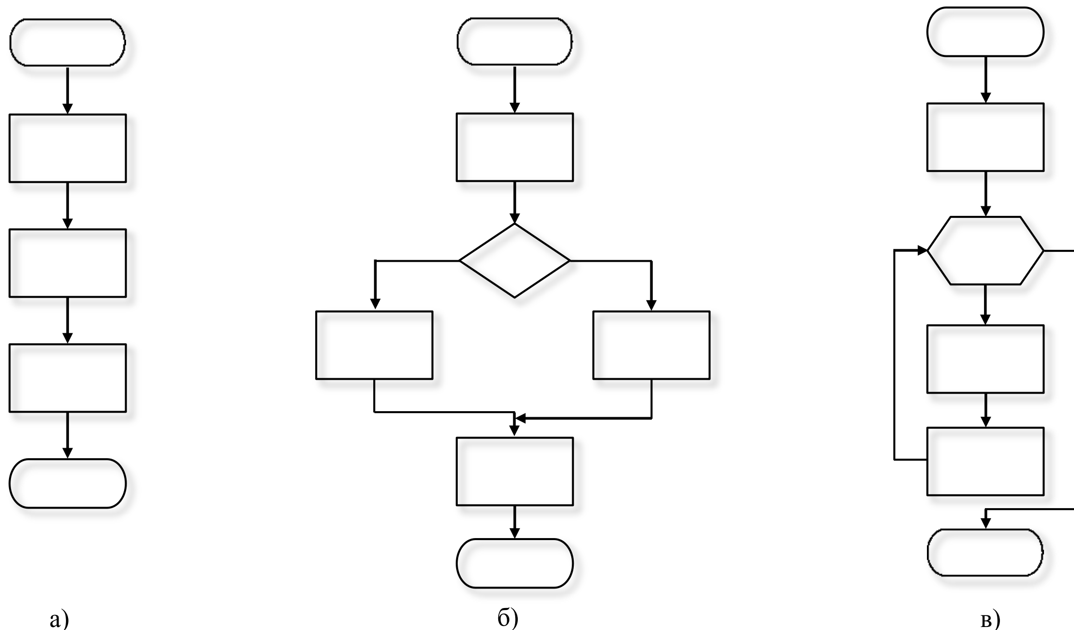


Рисунок 1.1 – Види алгоритмів

Цикл програми – це послідовність команд (тіло циклу), яка може виконуватися багаторазово (для нових початкових даних) до виконання певної умови.

У деяких випадках при наявності однакових послідовностей вказівок (команд) для різних даних з метою скорочення запису також виділяють *допоміжний алгоритм*.

Допоміжний алгоритм (процедура) – це алгоритм, який був розроблений раніше і цілком використовується при алгоритмізації конкретного завдання.

1.4.2. Структурне зображення алгоритму

На всіх етапах підготовки до алгоритмізації завдання широко використовується структурне зображення алгоритму.

Схема алгоритму (блок-схема алгоритму) – це графічне зображення алгоритму у вигляді схеми пов'язаних між собою за допомогою стрілок (ліній переходу) блоків – графічних символів, кожен з яких відповідає одному кроку алгоритму.

Усередині блоку дається опис відповідної дії. Графічне зображення алгоритму широко використовується перед програмуванням завдання внаслідок його наочності, оскільки зорове сприйняття зазвичай полегшує:

- процес написання програми;

- її коригування при можливих помилках; - осмислення процесу обробки інформації.

Правила зображення фігур наведені в міждержавному стандарті «ГОСТ 19.701-90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения».

Цей стандарт поширюється на умовні позначення (символи) в схемах алгоритмів, програм, даних і систем і встановлює правила виконання схем, які використовуються для відображення різних видів завдань обробки даних і засобів їх вирішення.

Стандарт не поширюється на форму записів і позначок, які розміщені всередині символів або поруч з ними та необхідні для уточнення виконуваних

ними функцій.

Загальні вимоги (обов’язкові):

1) Схеми алгоритмів, програм, даних і систем (далі – схеми) складаються з короткого пояснювального тексту і з’єднувальних ліній, які мають задане значення символів.

2) Схеми можуть використовуватися на різних рівнях деталізації, причому число рівнів залежить від розмірів і складності завдання обробки даних. Рівень деталізації повинен бути таким, щоб різні частини і взаємозв’язок між ними були зрозумілі в цілому.

У стандарті використовуються такі поняття:

Основний символ – символ, який використовується в тих випадках, коли точний тип (вид) процесу або носія даних невідомий або відсутня необхідність в описі фактичного носія даних.

Специфічний символ – символ, який використовується в тих випадках, коли відомий точний тип (вид) процесу або носія даних або коли необхідно описати фактичний носій даних.

Схема – це графічне зображення визначення, аналізу або методу розв’язання задачі, в якому використовуються символи для відображення операцій, даних, потоку, обладнання та інше.

Схема даних містить:

- символи даних (можуть відображати тип носія даних); - символи процесу, який потрібно виконати над даними;

- символи ліній, що вказують потоки даних між процесами та носіями даних;

- спеціальні символи (для зручності читання схеми).

Схема програми складається з:

- символів процесу, що вказують фактичні операції обробки даних (включаючи символи, що визначають шлях, якого слід дотримуватися з урахуванням логічних умов);

- лінійних символів, що вказують потік управління;

- спеціальних символів, які використовуються для полегшення написання та читання схеми.

Основні елементи схеми алгоритму

При зображенні елементів рекомендується дотримуватися строгих розмірів, визначених двома значеннями *a* і *b*. Значення a вибирається з ряду 15 мм, 20 мм, 25 мм, ... Значення b розраховується зі співвідношення *a* = 1,5 *b*. Визначення розмірів несе рекомендаційний характер, проте, варто відзначити, що при дотриманні даних розмірів блок-схеми мають більш акуратний вигляд.

Символ **«Дані»** є основним і відображає ввід/вивід даних (рис. 1.2). Він не визначає носія даних (для зазначення типу носія використовуються специфічні символи) і виконує перетворення даних у форму, яка придатна для обробки (вводу) або відображення результатів обробки (виводу).

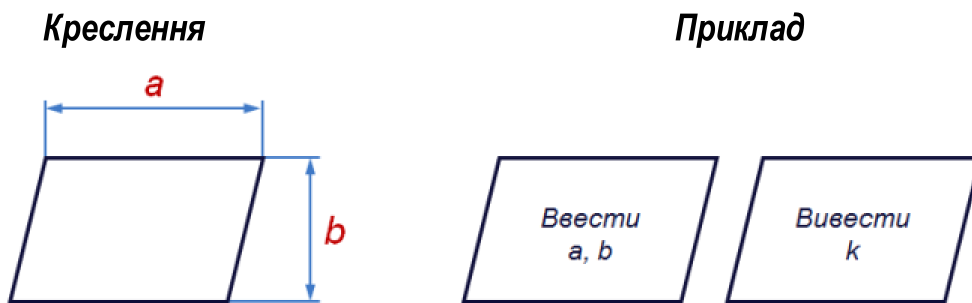


Рисунок 1.2 – Приклад використання символу «Дані»

Символ **«Ручний ввід»** (рис. 1.3) є специфічним і відображає дані, що вводяться вручну під час обробки з пристроїв будь-якого типу (клавіатура, перемикачі, кнопки, світлове перо, смужки зі штрих-кодом). В схемах програм не використовується.

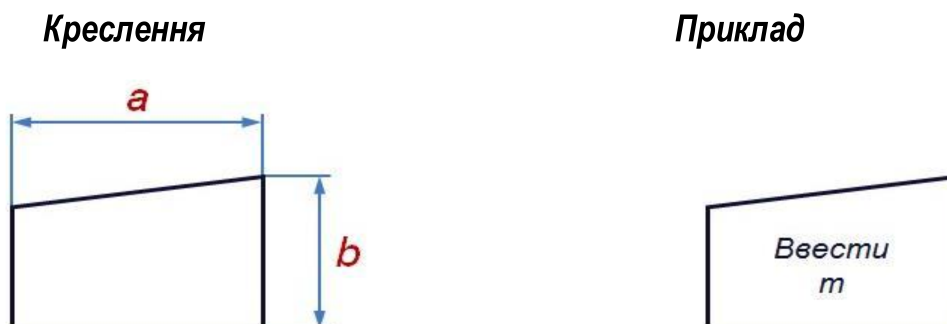


Рисунок 1.3 – Приклад використання символу «Ручний ввід»

Символ **«Процес»** відображає функцію обробки даних будь-якого виду (виконання певної операції або групи операцій, що призводить до зміни значення, форми або розміщення інформації або до визначення, за яким з декількох напрямків потоку слід рухатися). На рис. 1.4 представлені начертання символу «Процес» і приклад його використання.

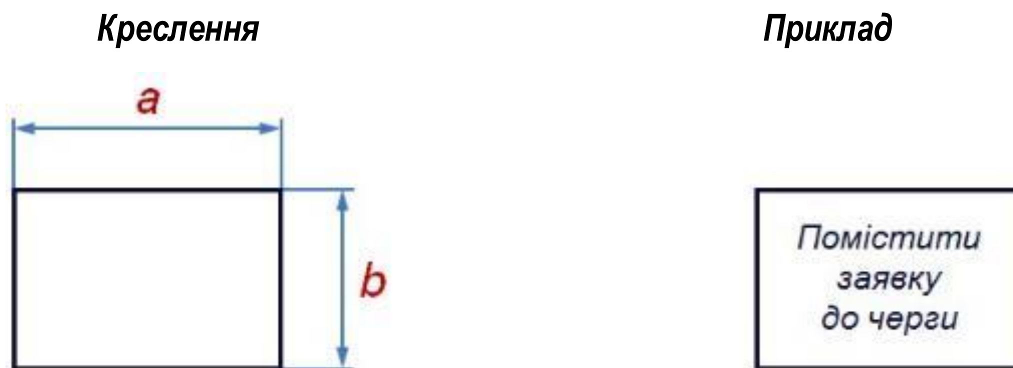


Рисунок 1.4 – Приклад використання символу «Процес»

Символ **«Зумовлений процес»** відображає зумовлений процес, що складається з однієї або декількох операцій або кроків програми, які визначені в іншому місці (в підпрограмі, модулі). Наприклад, в програмуванні – це виклик процедури або функції (рис. 1.5).

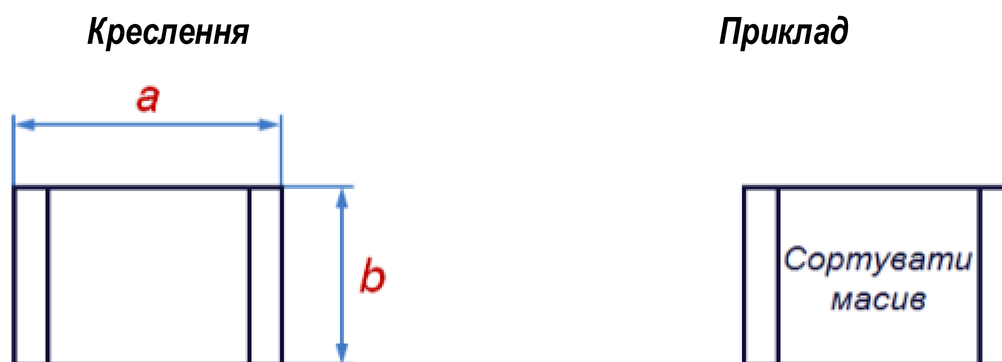


Рисунок 1.5 – Приклад використання символу «Зумовлений процес»

Символ ***«Розв'язання»*** відображає питання (умову) або функцію перемикача типу, що має один вхід і ряд альтернативних виходів, один і тільки один з яких може бути активізований після обчислення умов, визначених всередині цього символу (рис. 1.6). Вхід в елемент позначається лінією, що

входить зазвичай в верхню частину елемента. Якщо виходів два або три, то зазвичай кожен вихід позначається лінією, що виходить з решти вершин (бічних і нижньої). Якщо виходів більше трьох, то їх слід показувати однією лінією, що виходить з вершини (частіше нижньої) елемента, яка потім розгалужується. Відповідні результати обчислень можуть записуватися поряд з лініями, які відображають ці шляхи. Приклади розв'язання: у загальному випадку – порівняння (три виходи:>, <, =); в програмуванні – умовні оператори **if** (два виходи: **true**, **false**) і **case** (безліч виходів).

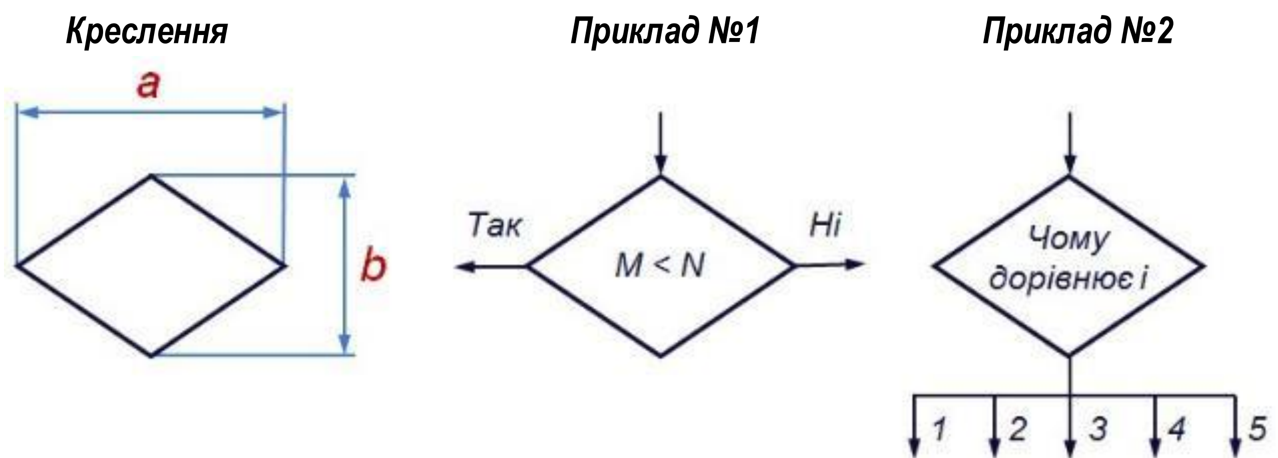


Рисунок 1.6 – Приклад використання символу «Розв’язання»

Символ **«Підготовка»** (рис. 1.7) відображає модифікацію команди або групи команд з метою впливу на деяку подальшу функцію (встановлення перемикача, модифікація індексного регістра або ініціалізація програми).

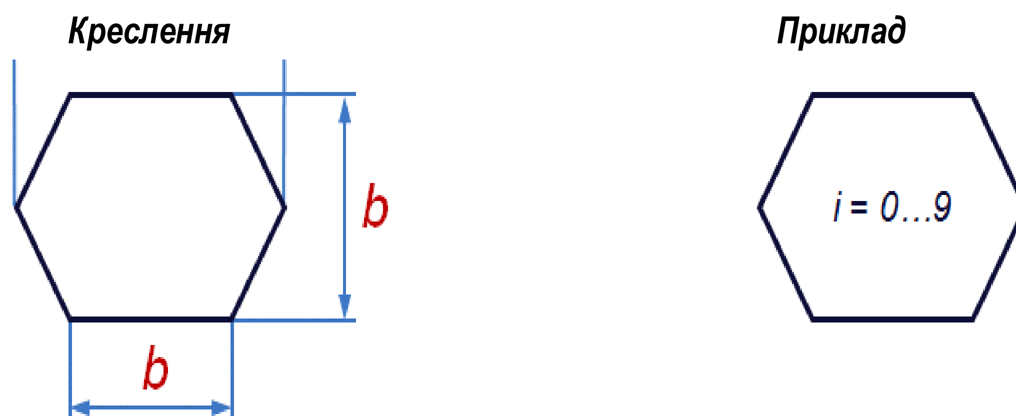


Рисунок 1.7 – Приклад використання символу «Підготовка»

Символ **«Межа циклу»,** що складається з двох частин, відображає початок і кінець циклу (рис. 1.8). Обидві частини символу мають однаковий ідентифікатор. Умови для ініціалізації, збільшення, завершення та інше поміщаються всередині символу на початку або в кінці в залежності від розташування операції, що перевіряє умову.

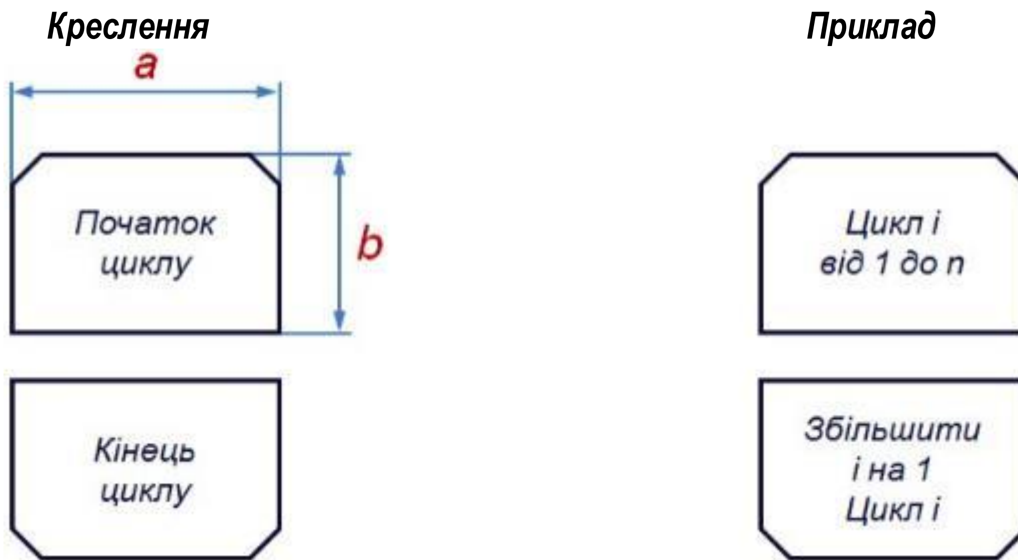


Рисунок 1.8 – Приклад використання символу «Межа циклу»

Символ **«Лінія»** відображає потік даних або управління. За необхідності або для підвищення зручності читання можуть бути додані стрілки-вказівники (рис. 1.9).

Рисунок 1.9 – Приклади використання символу «Лінія»

Символ **«Пунктирна лінія»** відображає альтернативний зв'язок між двома або більшою кількістю символів (рис. 1.10). Крім того, символ використовують для обведення анотованої ділянки.

Рисунок 1.10 – Приклади використання символу «Пунктирна лінія»

Символ **«З'єднувач»** (рис. 1.11) відображає вихід в частину схеми та вхід з іншої частини цієї схеми і використовується для обриву лінії і продовження її в іншому місці. Відповідні символи-з'єднувачі повинні містити одне й те ж саме унікальне позначення.

Рисунок 1.11 – Приклад використання символу «З'єднувач»

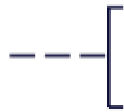
Символ **«Термінатор»** (рис. 1.12) відображає вхід із зовнішнього середовища і вихід у зовнішнє середовище (початок або кінець схеми програми, зовнішнє використання і джерело або пункт призначення даних).

Рисунок 1.12 – Приклад використання символу «Термінатор»

На практиці мають сенс такі описи термінаторів: початок / кінець, запуск / останов, перезапуск (мається на увазі перезапуск даної блок-схеми), помилка (мається на увазі завершення алгоритму з помилкою), вимкнення (мається на увазі виконання програмного вимкнення).

Символ **«Коментар»** (рис. 1.13) використовують для додавання описових коментарів або пояснювальних записів з метою пояснення або приміток. Пунктирні лінії в символі коментаря пов'язані з відповідним символом або можуть обводити групу символів. Текст коментарів або приміток повинен бути поміщений поряд з фігурою.

Креслення ***Приклад***



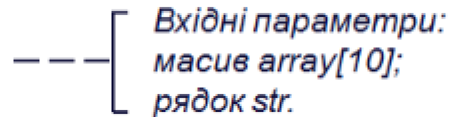


Рисунок 1.13 – Використання символу «Коментар»

Також символ коментаря слід використовувати в тих випадках, коли обсяг тексту, який вміщується всередині деякого символу (наприклад, символу процесу, символу даних та ін.), перевищує розмір самого символу. Коментарі використовують спільно з термінаторами для опису вхідних аргументів алгоритму при описі функцій.

Символ **«Пропуск»** (три крапки) використовують в схемах для відображення пропуску символу або групи символів, в яких не визначені ні тип, ні число символів (рис. 2.14). Символ використовують тільки в символах лінії або між ними. Він застосовується головним чином в схемах, що зображують спільні рішення з невідомим числом повторень.

Рисунок 1.14 – Приклади використання символу «Пропуск»

1.4.3. Правила виконання з’єднань

Потоки даних або потоки управління в схемах показуються лініями. Напрямок потоку зліва направо і зверху вниз вважається стандартним.

У випадках, коли необхідно внести більшу ясність в схему (наприклад, при з’єднаннях), на лініях використовуються стрілки. Якщо потік має відмінний від стандартного напрямок, стрілки повинні вказувати цей напрямок.

У схемах слід уникати перетину ліній. Лінії, що перетинаються, не мають логічного зв’язку між собою, тому зміни напрямку в точках перетину не допускаються (рис. 1.15).

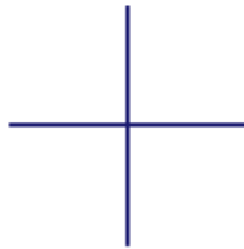


Рисунок 1.15 – Приклад перетину ліній

Дві або більше вхідні лінії можуть об’єднуватися в одну вихідну лінію. Якщо дві або більше лінії об’єднуються в одну лінію, місце об’єднання повинне бути зміщене (рис. 1.16).

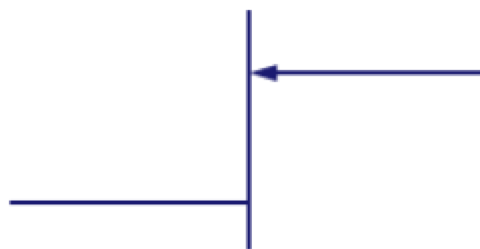


Рисунок 1.16 – Приклад об’єднання ліній

Лінії в схемах повинні підходити до символу або ліворуч, або зверху, а виходити або праворуч, або знизу. Лінії повинні бути спрямовані до центру символу. При необхідності лінії в схемах слід розривати, щоб уникнути зайвих перетинів або занадто довгих ліній, а також, якщо схема складається з декількох сторінок.

З’єднувач на початку розриву називається зовнішнім з’єднувачем, а з’єднувач в кінці розриву – внутрішнім з’єднувачем.

Посилання до сторінок можуть бути приведені спільно з символом коментаря для їх з’єднувачів.

1.4.4. Приклади використання схем алгоритмів

Приклад 1: Використання символів «Обмежувач» (рис. 1.17).

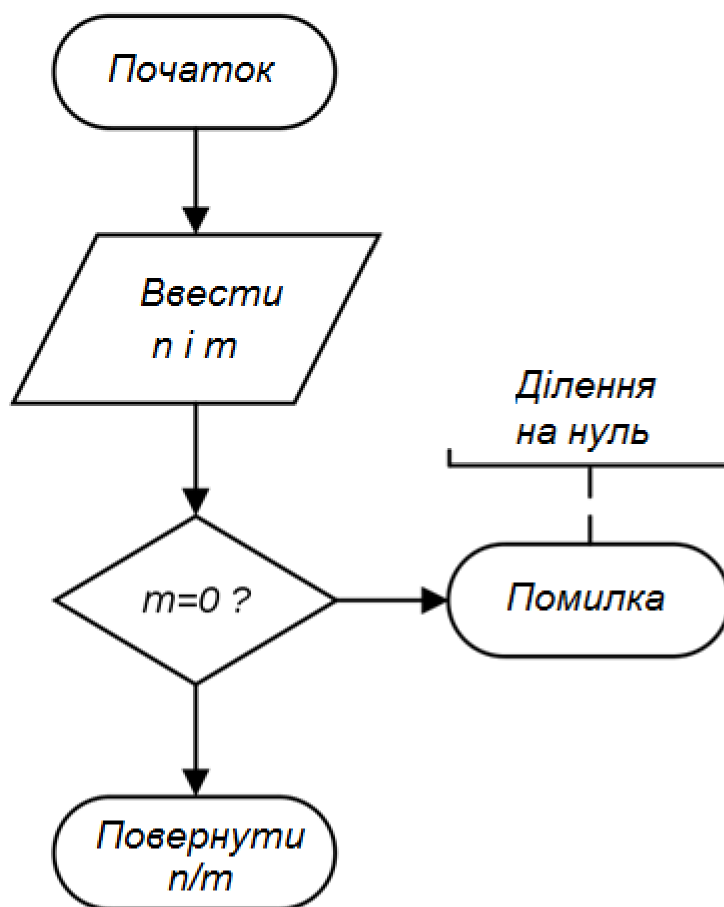


Рисунок 1.17 – Приклад використання символів «Обмежувач»

14

Приклад 2: Блок-схема алгоритму для розрахунку факторіала з використанням символів «Межа циклу» (рис. 1.18).

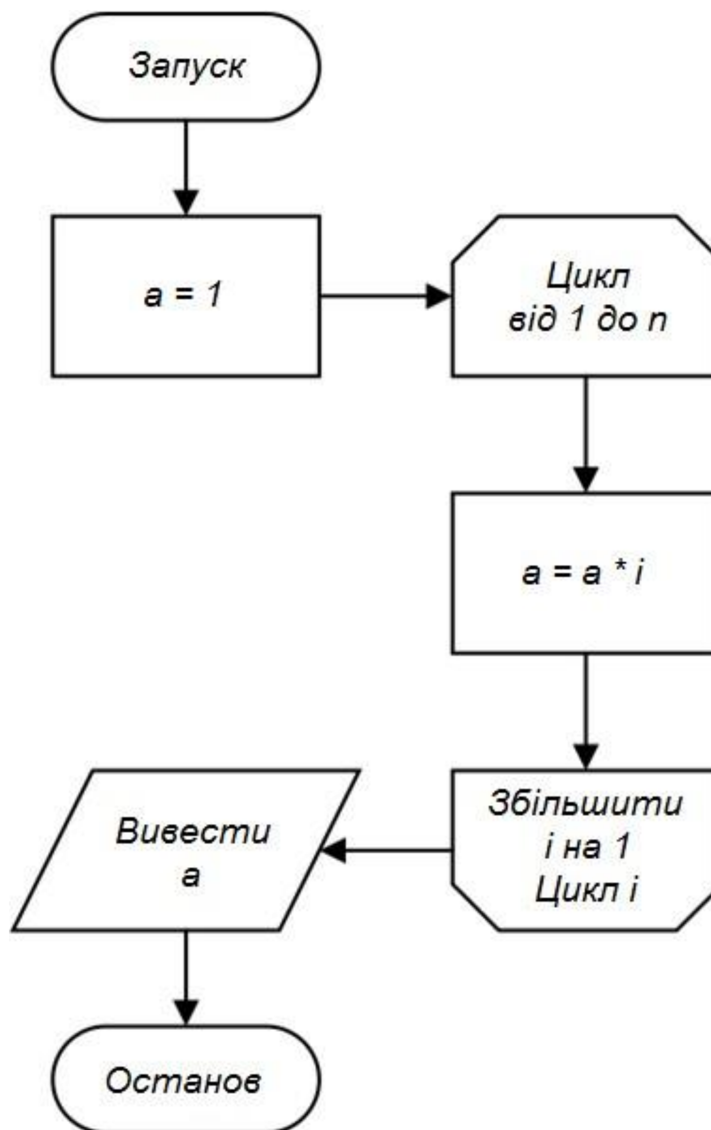


Рисунок 1.18 – Приклад використання символу «Межа циклу»

Приклад 3: Використання вкладених циклів (рис. 1.19).

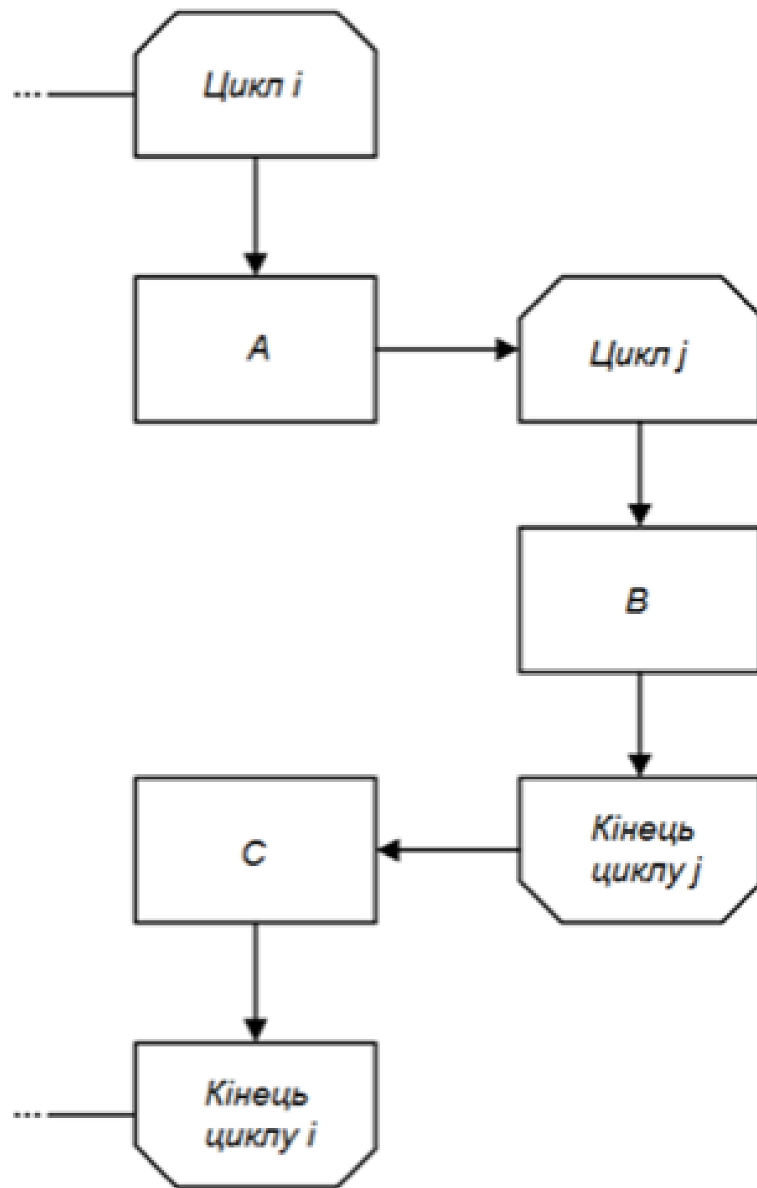


Рисунок 1.19 – Приклад використання вкладених циклів

Приклад 4: Розподіл алгоритму на дві частини з використанням з’єднувачів (рис. 1.20).

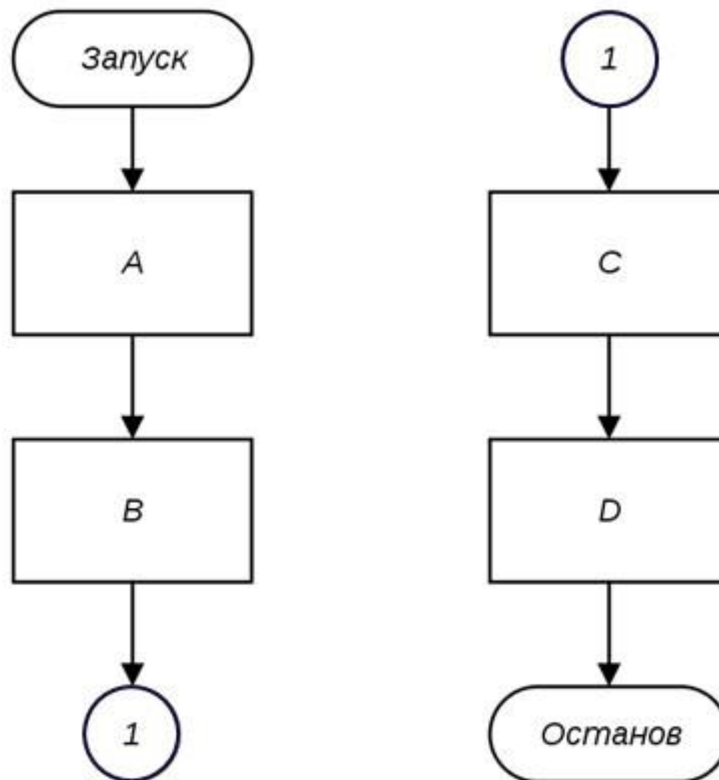


Рисунок 1.20 – Приклад використання з’єднувачів

Приклад 5: Використання коментарів (рис. 1.21).

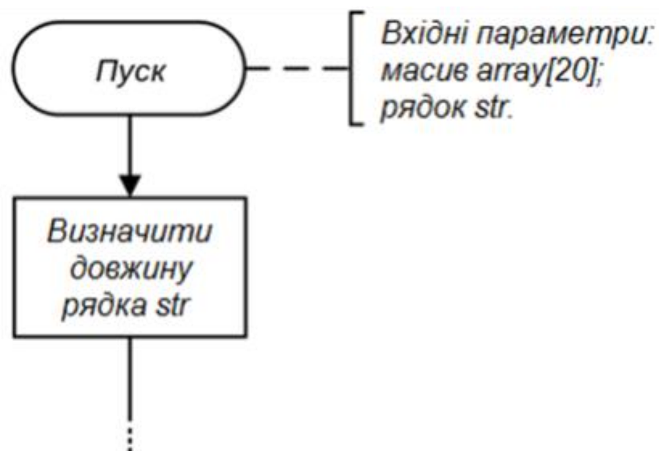


Рисунок 1.21 – Приклад використання коментарів

Контрольні запитання та завдання

1. Дайте визначення поняттю «алгоритм».
2. Перерахуйте основні властивості алгоритмів і розкрийте їх сутність.
3. Як поділяються алгоритми за типом реалізованого обчислювального процесу?
4. Які способи опису алгоритмів вам відомі?
5. Що розуміється під графічним способом опису алгоритмів? У чому полягає перевага даного способу перед словесним описом алгоритму?
6. Що таке трасування алгоритму? Для чого воно призначене?
7. Що розуміється під рівнем мови програмування?
8. Перерахуйте переваги та недоліки машинних мов.
9. Які основні етапи рішення завдань за допомогою комп'ютера?
10. Чим відрізняєтьс математичне моделювання від візуально-натурного моделювання?
11. Перелічіть основні етапи процесу розробки програм.
12. Як можна здійснити контроль тексту програми до виходу на комп'ютер?
13. Дайте визначення понятть «відлагодження» та «тестування».
14. У чому полягає суть відлагодження програм?
15. Наведить відміннсть тесту від тестування програми.
16. Що відноситься до тестових данних?
17. Перелічить етапи процесу тестування.
18. Що таке синтаксичні помилки?
19. Що відноситься до супровіду програм?
20. Які існують види алгоритмів ?
21. Що відноситься до основних елементів схеми алгоритму ?
22. Опишить правила виконааня зєднань.
23. Наведіть приклади використання схем алгоритмів.

Завдання для самостійного розв’язання

1. За допомогою лінійки побудуйте блок-схему лінійного алгоритму для формули $D = 9.8a^2 + 5.52\cos t^5$.

2. Побудуйте блок-схему розгалуженого алгоритму де перша гілка алгоритму (`true`) задана виразом $H = \sin y^2 - 2.8y + \sqrt{|y|}$, а друга гілка алгоритму (`false`) виразом $W = e^{t+r} + 7.2\sin r$.

3. Побудуйте блок-схему циклічного алгоритму з передумовою для виразу $y = 2^n$

4. Обчисліть значення y = n×m, використовуючи циклічний алгоритм з передумовою. Множення замінити додаванням n разів значення змінної m.

5. Обчисліть суму всіх парних чисел від 1 до 30, використовуючи циклічний алгоритм з передумовою.

6. Обчисліть добуток усіх непарних чисел від 1 до 10, використовуючи циклічний алгоритм з передумовою.

7. Обчисліть добуток усіх чисел від 1 до 15, які діляться на 3 без остачі, використовуючи циклічний алгоритм з передумовою.

8. Обчисліть суму всіх непарних чисел від 11 до 40, використовуючи циклічний алгоритм з передумовою.

9. Побудуйте блок-схему циклічного алгоритму з постумовою. (Обчисліть добуток усіх непарних чисел від 1 до 10.)

10. Обчисліть суму всіх чисел від 1 до 40, які діляться на 6 без остачі та побудуйте блок-схему циклічного алгоритму з постумовою.

11. Побудуйте блок-схему циклічного алгоритму із визначеною кількістю ітерацій для функції $y = \cos^2(2x) + x$, де початкове значення відвовідно дорівнює 1, крок - 0.2 та ітерацій -10 .

2. ОСНОВИ ПРОГРАМУВАННЯ

2.1. Вступ до програмування

Що таке програмування?

З одного боку. **Програмування** (як наука) – це розділ інформатики, галузь знань про алгоритми і програми та їх властивості, а також виконавців алгоритмів і програм.

Програмування містить у собі наступні розділи:

1) **Алгоритміка** – це наука про алгоритми та їх виконавців, яка дуже тісно пов'язана з математикою. На стику з математикою лежить математичне програмування.

2) **Системне програмування** – дисципліна, що пов'язана з написанням системного програмного забезпечення (ПЗ). Основна сфера – програми, що займаються обробкою даних і їх поданням.

3) **Прикладне програмування** – дисципліна, що пов'язана з написанням прикладного і, деякою мірою, інструментального програмного забезпечення.

Однією з основних відмінних рис системного програмування у порівнянні з прикладним є те, що результатом останнього є випуск програм для взаємодії з користувачем (наприклад, текстовий процесор). У той час як результатом системного програмування є випуск програм для взаємодії з апаратним забезпеченням (наприклад, дефрагментація жорсткого диска), що має на увазі сильну залежність таких програм від апаратної частини.

З іншого боку. **Програмування** – це процес і мистецтво створення комп'ютерних програм за допомогою мов програмування.

Програмування поєднує в собі елементи мистецтва, науки, математики та інженерії.

У вузькому сенсі слова, програмування розглядається як **кодування** – реалізація одного або декількох взаємопов'язаних алгоритмів на деякій мові програмування. *У більш широкому сенсі*, програмування – **процес**

створення програм, тобто розробка програмного забезпечення.

Програмування ґрунтується на використанні мов програмування, на ких записуються тексти програм. Велика частина роботи програміста пов'язана з написанням програмного коду на одній з мов програмування.

За висловом Ніклауса Вірта (створив такі мови як Паскаль, Модула-2, Оберон) **«Програми = алгоритми + структури даних»**.

Різні мови програмування підтримують різні стилі програмування (так звані «парадигми програмування»). Деякою мірою, мистецтво програмування полягає в тому, щоб вибрати одну з мов, яка найбільш повно підходить для вирішення наявного завдання. Різні мови вимагають від програміста різного рівня уваги до деталей при реалізації алгоритму, результатом чого часто буває компроміс між простотою та продуктивністю (або між часом програміста та часом користувача).

Рейтинги визначають мови програмування за різними параметрами. Топ **TIOBE** ґрунтується на кількості пошукових запитів, навчальних курсів і фахівців. Тут лідери виглядають так: Java**,** C, C++, Python, C#. Рейтинг IEEE Spectrum аналізує частоту пошукових запитів, число проектів на GitHub (соціальна мережа для розробників), а також згадки в Twitter і головних IT-порталах мережі. Тут в п'ятірці лідерів – Python, C++, Java, C, C#.

2.2. Сфери застосування програмування

Програмування має дуже багато сфер застосування, де технології набувають розвитку дуже швидкими темпами. Розробники потрібні у багатьох сферах, які навіть не завжди пов'язані тільки з інформаційними технологіями (**IT**). Зупинимося на найвідоміших і часто згадуваних експертами сферах.

2.2.1. Веб-розробка

Веб-розробка є одним з популярних і різнопланових напрямків. Тут працюють з такими мовами як JavaScript, PHP, Python, Javai Ruby, а також

використовують «мову структурованих запитів» SQL.

Безсумнівний плюс цієї сфери застосування програмування – досить легкий поріг входу, швидкість вивчення бібліотек і інструментів. Цим продиктована популярність Інтернету у новачків у програмуванні. Але й конкуренція тут є дуже високою: щоб залишатися на плаву, потрібно постійно стежити за тенденціями.

2.2.2. Мобільна розробка

Цей напрямок сьогодні вважають найпопулярнішим і перспективним. Смартфони є майже у всіх, і їх можливості безперервно зростають. Мови створення мобільних додатків: Java і Kotlin для Android, Swift для Apple, а також Python, JavaScript, C#.

Варто зазначити, що самостійне створення мобільних додатків високого доходу не принесе: успіх мають стартапи з унікальною і якісної ідеєю. Але завдяки самостійній розробці ви придбаєте необхідний досвід для роботи в команді.

2.2.3. Десктопні додатки

Про десктоп (**desktop**) говорять дещо менше й рідше, ніж про мобільні та веб-технології. Пов'язано це з тим, що всі питання в цій сфері давно вже вивчені, відповіді на них стандартні й зрозумілі, а зміни не відбуваються так швидко, як в інших нішах.

Потрапити до розробки додатків для робочого столу складніше, ніж до веб та мобільних розробок, – без вищої профільної освіти навряд чи візьмуть будь-кого в серйозні організації. Мови десктопу залежать від операційної системи:

- для **Linux** і міжплатформних додатків – **C++**;
- для **macOS** – **Swift** і **Objective-C**;
- для **Windows** – **C#**.

2.2.4. Розробка ігор

Не так давно розробка ігор була на піку зростання популярності. Сьогодні пристрасті дещо згасли, але це як і раніше шанована й цікава сфера інтернет-технологій.

Усвідомлення того, що твоїм продуктом користуються мільйони фанатів по всьому світу, дарує приголомшливу емоційну віддачу розробнику ігор. В індустрії отримують високі зарплати, але йти туди потрібно з готовністю до складної роботи та високих навантажень. Мови для розробки ігор: **C++**, **C#**, **Lua** і **JavaScript** для браузерних ігор.

2.2.5 Data Science

Data Science – це наука про методи аналізу даних і отримання з них цінної інформації, знань. Вона тісно перетинається з такими сферами як машинне навчання (**Machine Learning**) і науку про мислення (**Cognitive Science**) і, звичайно ж, технологіями для роботи з великими даними (**Big Data**).

За час масового поширення технологій людина згенерувала величезну кількість даних, яку вона не здатна обробити та візуалізувати. Дані про наші дзвінки і переміщення, поведінку в Інтернеті, переваги у магазинах, антропогенні зміни в ландшафті, кліматичні процеси і багато інших речей: це все – великі дані, **Big Data**. З них (за умов правильної обробки) можна отримати велику користь.

Для ефективної роботи з великими даними застосовують машинне навчання. У цьому випадку людина тільки дає комп'ютеру якісь початкові дані, але результати роботи такого алгоритму не визначаються людиною. Вона лише обирає спосіб навчання машини. Далі машина сама вчиться, приходить до тих чи інших відповідей і аналізує інформацію. Це схоже на те, як вчимося ми з вами. Машинне навчання – це не тільки штучний інтелект. До цієї сфери належать генетичні та еволюційні алгоритми, і більш прості завдання, які пов'язані, наприклад, з кластерним аналізом (багатовимірна статистична

процедура, що виконує збір даних, які містять інформацію про вибірку об'єктів, і потім впорядковує об'єкти в порівняно однорідні групи).

Cognitive Science – це міждисциплінарна наука, що вивчає, механізми пізнання та мислення. Результати таких досліджень в першу чергу лягають в основу розробки різних підходів до створення штучного інтелекту.

Необхідно відзначити, що **Data Science** знаходиться на стику інтернет-технологій і бізнесу.

Фахівцеві з **Big Data** необхідні серйозні знання математичного аналізу, статистики, машинного та глибокого навчання й текстової аналітики. Мови програмування, на яких «говорять» в цій сфері, – **R**, **SAS** і **Python**.

2.2.6.Програмування вбудованих систем

Embedded [ɪm'bedɪd], [em'bedɪd] – це мікроконтролери, промислове обладнання з числовим програмним керуванням (ЧПК) та ін. Тут потрібно добре розуміти апаратну частину машини, для якої створюється програмне забезпечення (ПЗ). Необхідні мови – **C**, **C++** і спеціалізовані мови для тих чи інших мікроконтролерів.

Потрапити в цю сферу дуже важко: і вакансій, і фахівців в ній набагато менше, ніж в тому ж Інтернеті. Але якщо вивчати технології, які пов'язані з використанням в пристроях Інтернету, можна піти далі.

2.2.7. Інтернет речей (Internet of Things, IoT)

Інтернет речей частково є сегментом ринку, який перетинається з програмуванням вбудованих систем. Інтернету речей пророкують велике майбутнє й активний розвиток найближчим часом.

Інтернет речей – це створення **smart**-пристроїв, які підключені до мережі «розумного» міста або будинку. Напрямок молодий і перспективний, тому увійти в нього сьогодні простіше, ніж через 10-20 років.

Необхідні мови: **Java, C, JavaScript, Python, C++**.

2.2.8. Хмарні обчислення

Програмні продукти для компаній із трендів не підуть: навпаки, з'являються нові і конкурують між собою. Найбільш популярними на сьогодні є хмарні технології.

Хмарні обчислення (англ. **Cloud computing**) – це модель забезпечення зручного мережевого доступу за вимогою до деякого загального фонду обчислювальних ресурсів (наприклад, мереж передачі даних, серверів, пристроїв зберігання даних, додатків і сервісів – як разом, так і окремо), які конфігуруються та можуть бути оперативно надані й звільнені з мінімальними експлуатаційними витратами або зверненнями до провайдера.

До моделей обслуговування хмарних обчислень відносять:

- **SaaS** – рішення для менеджерів;
- **PaaS** – ПЗ для розробників;
- **IaaS** – мережеві ресурси в якості віртуальних машин і зберігання даних.

SaaS (англ. **Software as a service** – програмне забезпечення як послуга) – це одна з форм хмарних обчислень, модель обслуговування, при якій передплатникам надається готове прикладне програмне забезпечення, як повністю обслуговується провайдером. Постачальник в цій моделі самостійно керує додатком, надаючи замовникам доступ до функцій з клієнтських пристроїв, як правило, через мобільний додаток або веб-браузер.

Приклади: G Suite [swiːt] (Google Apps), Microsoft Office 365.

Служба **G Suite** підтримує кілька веб-додатків зі схожою функціональністю як у традиційних офісних пакетів, і включає: **Gmail, Google Calendar, Google Talk, Google Docsi Google Sites.**

Частіше за інших, в якості прикладів використання SaaS-рішень, можна зустріти системи керування проектами, і спільної роботи над ними, онлайнові органайзери, системи документообігу.

Основна перевага моделі **SaaS** для споживача послуги полягає у відсутності витрат, що пов'язані зі встановленням, оновленням і підтримкою працездатності обладнання та працюючого на ньому програмного забезпечення.

У моделі **SaaS**:

- додаток пристосований для віддаленого використання; - одним додатком користується декілька клієнтів;

- оплата стягується або у вигляді щомісячної абонентської плати, або на основі обсягу операцій;

- технічна підтримка програми включена в оплату;

- модернізація та оновлення додатка відбувається оперативно і прозоро для клієнтів.

Як і у всіх формах хмарних обчислень, замовники платять не за володіння програмним забезпеченням як таким, а за його оренду (тобто за його використання через мобільний додаток або веб-інтерфейс). Таким чином, на відміну від класичної схеми ліцензування програмного забезпечення, замовник несе порівняно невеликі періодичні витрати, і йому не потрібно інвестувати значні кошти в придбання прикладної програми та необхідних програмно-платформних і апаратних засобів для його розгортання, а потім підтримувати його працездатність. Схема періодичної оплати припускає, що якщо необхідність в ПЗ тимчасово відсутня, то замовник може призупинити його використання і заморозити виплати розробнику.

З точки зору розробника деякого пропрієтарного програмного забезпечення (ПЗ, що є приватною власністю авторів або правовласників і не задовольняє критеріям вільного ПЗ (наявності відкритого програмного коду недостатньо)) модель **SaaS** дозволяє ефективно боротися з неліцензійним програмним забезпеченням, оскільки ПЗ як таке не потрапляє до кінцевих замовників. Крім того, концепція **SaaS** часто дозволяє зменшити витрати на розгортання й впровадження систем технічної і консультаційної підтримки продукту, хоча й не виключає їх повністю.

В обов’язки програмістів входить впровадження та оновлення системи, а також навчання працюючих з нею співробітників. Але є й фахівці, які розробляють ці оновлення, пишуть і підтримують код.

У сфері **SaaS** відносно невисокий поріг входу й непогана заробітна плата, є можливість розвиватися і як програміст, і як фінансист.

Platform as a Service (**PaaS**, «платформа як послуга») – модель надання хмарних обчислень, при якій споживач отримує доступ до використання інформаційно-технологічних платформ: операційних систем, систем керування базами даних, програмного забезпечення, засобів розробки та тестування, які розміщені у хмарного провайдера. В цій моделі вся інформаційно-технологічна інфраструктура, включаючи обчислювальні мережі, сервери, системи зберігання, цілком керується провайдером. Провайдером також визначається набір доступних для споживачів видів платформ і набір керованих параметрів платформ, а споживачеві надається можливість використовувати платформи, створювати їх віртуальні екземпляри, встановлювати, розробляти, тестувати, експлуатувати на них прикладне програмне забезпечення, при цьому динамічно змінюючи кількість споживаних обчислювальних ресурсів.

Приклади: AWS Elastic Beanstalk, Windows Azure,Heroku, Force.com, Google App Engine, Apache Stratos.

Інфраструктура як послуга (англ. **Infrastructure as a Service; IaaS**) – одна з моделей обслуговування в хмарних обчисленнях, за якою споживачам надаються за передплатою фундаментальні інформаційно-технологічні ресурси – віртуальні сервери із заданою обчислювальною потужністю, операційною системою (найчастіше – встановленою провайдером з шаблону) і доступом до мережі.

Приклади: Amazon EC2, Windows Azure, Rackspace, Google Compute Engine.

2.3. Парадигми програмування

В основі тієї чи іншої мови програмування лежить певна керівна ідея, що істотно впливає на стиль відповідних програм. Залежно від призначення та/або способу написання програм розрізняють **парадигми** (також відомі як

підходи або **технології) програмування**.

Парадигма програмування – це сукупність принципів, методів і понять, що визначають спосіб конструювання програм.

Парадигма визначається:

- обчислювальної моделлю;

- базової програмної одиницею або одиницями;

- методами поділу абстракцій.

Мова програмування не обов’язково використовує тільки одну парадигму. Мультипарадигмальними називають мови, які підтримують декілька парадигм. Творці таких мов дотримуються точки зору, згідно з якою жодна парадигма не може бути однаково ефективною для всіх завдань, і слід дозволяти програмісту обирати кращий стиль програмування для вирішення кожного окремого завдання. Своїм сучасним значенням в науково-технічній галузі термін «парадигма» зобов’язаний, мабуть, Томасу Куну та його книзі «Структура наукових революцій». Кун називав парадигмами усталені системи наукових поглядів, в рамках яких ведуться дослідження. Згідно з Куном, в процесі розвитку наукової дисципліни може статися заміна однієї парадигми на іншу. При цьому стара парадигма ще продовжує деякий час існувати й навіть розвиватися завдяки тому, що багато її прихильники виявляються з тих чи інших причин не здатними перебудуватися для роботи в іншій парадигмі.

Термін «парадигма програмування» вперше застосував у 1978 році Роберт Флойд у своїй лекції лауреата премії Тюрінга. Флойд відзначав, що в програмуванні можна спостерігати явище, подібне парадигм Куна, але, на відміну від них, парадигми програмування не є взаємовиключними: якщо прогрес мистецтва програмування в цілому вимагає постійного винаходу і вдосконалення парадигм, то вдосконалення мистецтва окремого програміста вимагає, щоб він розширював свій репертуар парадигм. На думку Роберта Флойда, на відміну від парадигм в науковому світі, описаних Куном, парадигми програмування можуть поєднуватися, збагачуючи інструментарій програміста.

Парадигма програмування не визначається однозначно мовою програмування. Практично всі сучасні мови програмування тією чи іншою мірою допускають використання різних парадигм.

Так на мові `C`, яка не є об'єктно-орієнтованою, можна працювати відповідно до принципів об'єктно-орієнтованого програмування, хоча це й пов'язано з певними складнощами. Функціональне програмування можна застосовувати при роботі на будь-якій імперативній мові, в якій є функції (для цього достатньо не застосовувати присвоювання), та інше.

2.3.1. Імперативне програмування

Імперативне програмування – це парадигма програмування, для якої характерним є наступне:

- у початковому коді програми записуються інструкції (команди); - інструкції повинні виконуватися послідовно;

- дані, які отримані при виконанні попередніх інструкцій, можуть читатися з пам'яті наступними інструкціями;

- дані, які отримані при виконанні інструкції, можуть записуватися в пам'ять.

Імперативна програма схожа на накази (англ. **`Imperative`** – наказ, наказовий спосіб), що виражаються наказовим способом в природних мовах, тобто представляють собою послідовність команд, які повинен виконати комп'ютер.

При імперативному підході до складання коду (на відміну від функціонального підходу, що відноситься до декларативної парадигми) широко використовується присвоювання. Наявність операторів присвоювання збільшує складність моделі обчислень і робить імперативні програми схильними до специфічних помилок, які не зустрічається при функціональному підході.

Основні риси імперативних мов:

- використання іменованих змінних;

- використання оператора присвоювання; - використання складових виразів;
- використання підпрограм та інше.

Мови, що підтримують дану парадигму:

Як основну: **Assembler**, **Fortran, Algol, Cobol, Pascal, C, C++, Ada**.

Як допоміжну: **Python, Ruby, Java, C#, PHP, Haskell** (через монади).

2.3.2. Декларативне програмування

Декларативне програмування – це парадигма програмування, в якій задається специфікація рішення задачі, тобто описується, що представляє собою проблема й очікуваний результат.

Є протилежністю імперативного програмування.

Декларативні програми не використовують поняття стану, тобто не містять змінних і операторів присвоювання.

Мови, що підтримують дану парадигму: **SQL, HTML.**

Найбільш близьким до «чисто декларативного» програмування є написання виконуваних специфікацій. У цьому випадку програма являє собою формальну теорію, а її виконання є одночасно автоматичним доказом цієї теорії.

2.3.3. Процедурне (алгоритмічне) програмування

Процедурне програмування – це програмування на імперативній мові, при якому оператори, що виконуються послідовно, можна зібрати в підпрограми за допомогою механізмів самої мови.

Процедурне програмування є відображенням архітектури традиційних ЕОМ, яка була запропонована Фон Нейманом у 1940-х роках. Теоретичною моделлю процедурного програмування слугує абстрактна обчислювальна система під назвою машина Тюрінга.

Архітектура фон Неймана (модель фон Неймана,Принстонська архітектура) – широко відомий принцип спільного зберігання команд і даних

у пам’яті комп’ютера (рис. 2.1), де АЛП – арифметико-логічний пристрій; ПК – пристрій керування.

Для звернення до цієї пам’яті використовується загальна системна шина, якою до процесору надходять і команди, і дані.

Рисунок 2.1 – Архітектура фон Неймана

Машина Тюрінга (МТ) – це абстрактний виконавець (абстрактна обчислювальна машина). Була запропонована Аланом Тюрінгом у 1936 році для формалізації поняття алгоритму.

Машина Тюрінга здатна імітувати всіх виконавців (за допомогою задавання правил переходу), що певним чином реалізують процес покрокового обчислення, в якому кожен крок обчислення досить простий.

Тобто, будь-який інтуїтивний алгоритм може бути реалізований за допомогою деякої машини Тюрінга.

До складу машини Тюрінга входить необмежена в обидва боки стрічка (можливі машини Тюрінга, які мають кілька нескінченних стрічок), розділена на осередки, і пристрій керування (також називається головкою запису-читання (ГЗЧ)), здатний перебувати в одному стані з нескінченної кількості станів. Кількість можливих станів пристрою керування скінчена і точно задана.

Пристрій керування може переміщатися вліво і вправо по стрічці, читати і записувати в комірки символи деякого скінченого алфавіту. Виділяється особливий порожній символ, що заповнює всі комірки стрічки, крім тих з

них (скінченого числа), на яких записані вхідні дані.

Пристрій керування працює згідно з правилами переходу, які представляють алгоритм, реалізований цією машиною Тюрінга. Кожне правило переходу наказує машині, в залежності від поточного стану і спостережуваного в поточній комірці символу, записати в цю комірку новий символ, перейти до нового стану і переміститися на одну комірку вліво або вправо. Деякі стани машини Тюрінга можуть бути позначені як термінальні, і перехід до будь-якого з них означає кінець роботи, зупинку алгоритму.

Мови, що підтримують дану парадигму:

Як основну: **C, C++, Pascal, Object Pascal**.

Як допоміжну: **C#, Java, Ruby, Python, JavaScript**. Підтримують частково: ранній **Basic**.

2.3.4. Структурне програмування

Структурне програмування – методологія програмування, що базується на системному підході до аналізу, проектування та реалізації програмного забезпечення. Ця методологія народилася на початку 70-х років і виявилася настільки життєздатною, що й до сих пір є основною у великій кількості проектів. Її основу складають такі положення:

- *складне завдання розбивається на більш дрібні завдання*, які є функціонально краще керованими. Кожне завдання має один вхід і один вихід. У цьому випадку керуючий потік програми складається з сукупності елементарних підзадач з ясним функціональним призначенням;

- *простота керуючих структур, які використовуються у завданні*. Це положення означає, що логічно завдання повинне складатися з мінімальної, функціонально повної сукупності досить простих керуючих структур. Як приклад такої системи можна привести алгебру логіки, в якій кожна функція може бути виражена через функціонально повну систему: диз'юнкцію, кон'юнкцію і заперечення;

- *розробка програми повинна вестися поетапно*. На кожному етапі

має вирішуватися обмежене число чітко поставлених завдань з ясним розумінням їх значення і ролі в контексті всієї задачі. Якщо такого розуміння не досягається, це говорить про те, що даний етап надто великий і його потрібно розділити на більш прості кроки.

Мови, що підтримують дану парадигму: Як основну: **C, Pascal, Basic**.

Як допоміжну: **C#, Java, Python, Ruby, JavaScript**.

2.3.5. Аспектно-орієнтоване програмування

Аспектно-орієнтоване програмування (АОП) – парадигма програмування, заснована на ідеї поділу функціональності для поліпшення розбиття програми на модулі.

Методологія АОП була запропонована групою інженерів дослідницького центру Xerox PARCпід керівництвом Грегора Кічалеса (Gregor Kiczales). Ними ж було розроблено аспектно-орієнтоване розширення для мови Java, що отримало назву AspectJ– (2001 рік).

Основні поняття АОП:

Аспект (англ. **Aspect**) – модуль або клас, який реалізує наскрізну функціональність. Аспект змінює поведінку іншого коду, застосовуючи пораду в точках з’єднання, які визначені деякими зрізом.

Порада (англ. **Advice**) – засіб оформлення коду, який повинен бути викликаним з точки з'єднання. Порада може бути виконана до, після або замість точки з’єднання.

Точка з’єднання (англ. **Join point**) – точка у виконуваній програмі, де слід застосувати пораду. Багато реалізації АОП дозволяють використовувати виклики методів і звернення до полів об’єкта в якості точок з’єднання.

Зріз (англ. **Pointcut**) – набір точок з’єднання. Зріз визначає, чи підходить данаточка з’єднання до даної поради.

Найзручніші реалізації АОП використовують для визначення

зрізів синтаксис основної мови (наприклад, в **`AspectJ`** застосовуються **`Java`**-сигнатури) і допускають їх повторне використання за допомогою перейменування та комбінування.

Впровадження (англ. **`Introduction`**, введення) – зміна структури класу та/або зміна ієрархії успадкування для додавання функціональності аспекту в чужорідний код. Зазвичай реалізується за допомогою деякого метаоб'єктного протоколу (англ. **`Metaobject protocol, MOP`**).

Мова аспектно-орієнтованого програмування: **`AspectJ`**–аспектно-орієнтоване розширення мови **`Java`**.

2.3.6. Об'єктно-орієнтоване програмування (ООП)

Ідея ООП полягає в прагненні зв'язати дані з процедурами, що обробляють ці дані, в єдине ціле – об'єкт. ООП базується на трьох найважливіших принципах, які надають об'єктам нові властивості. Цими принципами є *інкапсуляція*, *успадкування* та *поліморфізм*.

Інкапсуляція – це об'єднання в єдине ціле даних і алгоритмів обробки цих даних. В рамках ООП дані називаються полями об'єкта, а алгоритми – об'єктними методами.

Успадкування – властивість об'єктів породжувати своїх потомків. Об'єкт – це потомок, що автоматично успадковує від батьків усі поля та методи, може доповнювати об'єкти новими полями та замінювати (перекривати) методи батька або доповнювати їх.

Поліморфізм – властивість споріднених об'єктів (тобто об'єктів, що мають одного загального батька) вирішувати схожі за змістом проблеми різними способами.

Мови ООП: **`C++, C#, Java, Python`** та інші.

2.3.7. Функціональне програмування

Функціональне програмування – парадигма програмування, в якій процес обчислення трактується як обчислення значень функцій в

математичному розумінні (на відміну від функцій як підпрограм у процедурному програмуванні).

При необхідності, в функціональному програмуванні вся сукупність послідовних станів обчислювального процесу подається явно, наприклад, списком.

Функціональне програмування передбачає обходитися обчисленням результатів функцій від початкових даних і результатів інших функцій, і не передбачає явного зберігання стану програми. Відповідно, не передбачає воно і змінність цього стану.

У функціональній мові при виконанні функції з одними й тими ж самими аргументами ми завжди отримаємо однаковий результат: вихідні дані залежать тільки від вхідних. Це дозволяє середовищам виконання програм на функціональних мовами кешувати результати функцій і викликати їх в порядку, що не визначається алгоритмом і распаралелювати їх без будь-яких додаткових дій з боку програміста (що забезпечують функції без побічних ефектів – чисті функції).

Лямбда-числення є основою для функціонального програмування. Багато функціональних мов можна розглядати як «надбудову» над ним.

Мови функціонального програмування: Lisp, Haskellта інші.

2.3.8. Логічне програмування

Логічне програмування – парадигма програмування, яка заснована на автоматичному доказі теорем. Логічне програмування засноване на теорії та апараті математичної логіки з використанням математичних принципів резолюцій.

Найвідомішою мовою логічного програмування є `Prolog`.

2.3.9. Узагальнене програмування

Узагальнене програмування (англ. **`Generic programming`**) – парадигма програмування, яка полягає в такому описі даних і алгоритмів,

що можна застосовувати до різних типів даних, не змінюючи саме цей опис. У тому чи іншому вигляді підтримується різними мовами програмування. Можливості узагальненого програмування вперше з'явилися у вигляді дженериків (узагальнених функцій) у 1970-х роках в мовах Клу і Ада, потім у вигляді параметричного поліморфізму в **ML** і його потомків, а потім у багатьох об'єктно-орієнтованих мовах, таких як **C++, Java, Object Pascal, D, Eiffel**, мовами для платформи **.NET** і інших.

2.4. Інструменти для ефективної роботи програміста

Робота програміста є досить складною, тому будь-який інструмент, який її полегшує, слід тільки вітати. Теоретично писати коди програм можна, маючи під рукою лише блокнот і компілятор, але на практиці програміст користується спеціальними інструментами для прискорення роботи.

Розглянемо сім інструментів програміста, які потрібні для повсякденної роботи.

2.4.1. Інтегроване середовище розробки

Інтегроване середовище розробки (**IDE, Integrated Development Environment**) – це основний інструмент, який програміст запускає відразу, як тільки приходить на роботу.

Типове середовище включає:

- редактор з підсвічуванням коду; - компілятор;

- відлагоджувач;

- керування проектами.

Існують універсальні **IDE**, які підтримують багато мов програмування (**Code::Blocks, NetBeans, Eclipse, Qt Creator, Geany**), а також спеціалізовані **IDE**, які націлені на одну мову програмування (**Visual Basic, Delphi, Dev-C++**).

На жаль, практика роботи програміста є такою, що знання однієї мови програмування й одного середовища програмування є явно недостатнім.

Постійно доводиться використовувати різні мови або середовища в залежності від завдання. Тому рекомендується пробувати й вивчати різні **IDE**.

В якості основного **IDE** краще вибрати те середовище розробки, яке для вас є найбільш комфортним і зручним, та вже його вивчити досконально. Це істотно підвищить вашу продуктивність.

2.4.2. Профілювальник коду (профайлер)

Профілювальник коду, або як часто його називають **профайлер** (від слова **profiler**) – це інструмент, який збирає інформацію про роботу програми. Як правило, профайлер потрібний в тих випадках, коли ваша програма працює не так швидко, як хотілося б.

Щоб знайти вузьке місце програми запускають профайлер, який фіксує час виконання різних фрагментів програми.

Існує багато профайлерів, як універсальних, так і спеціалізованих: **GlowCode** (**Windows**), **gprof** (**Linux/Unix**), **Intel Advisor** (**Linux** і **Windows**).

2.4.3. Система контролю версій

Часто буває так, що програміст вніс правки до початкового коду, і програма перестала працювати. Для швидкого поверненнядо працюючої версії використовуються системи контролю версій. Вони ведуть облік змін у файлах і дозволяють відкинути редагування до потрібної точки.

Найбільш популярними системами контролю версій (СКВ) є: **Subversion(SVN)**,**Git** і **Mercurial**.

Раніше СКВ працювали на комп'ютері розробника, але останнім часом використовуються веб-сервери для СКВ. Це дає багато переваг, оскільки можна працювати на різних комп'ютерах і мати копію програми у хмарі.

Істотних відмінностей між цими системами немає. Базовий функціонал однаковий, різниця лише в способах реалізації. Тому можете обрати ту систему, яка вам більше сподобається.

Найпопулярнішими серверами для СКВ є: **GitHub** (сервіс платний, але безкоштовний для проектів з відкритим вихідним кодом) і **Bitbucket** (безкоштовний сервіс).

2.4.4. Візуальний редактор інтерфейсу

Хоча інтерфейс програми можна повністю написати в коді, але це підходить тільки для зовсім простих програм. Програмування інтерфейсу вручну – це надзвичайно довгий і трудомісткий процес.

Є ще одна причина, щоб цей процес сильно прискорити – спілкування з замовником. Найперша розмова з замовником стане набагато продуктивнішою, якщо ви зможете швидко зробити макет програми. Замовники нічого не розуміють в програмуванні, але інтерфейс розуміють всі. Чим швидше ви покажете макет майбутньої програми, тим раніше отримаєте замовлення.

Існує багато редакторів інтерфейсу, які допомагають зробити зовнішній вигляд програми простим перетягуванням віджетів (**GUI**-конструкторів). Вони можуть бути як окремими програмами, наприклад, **Glade**, так і плагінами до **IDE**, як, наприклад, **wxSmith** для **Code::Blocks**.

Як правило, завдання візуального редактора – задати розташування елементів інтерфейсу, а код обробки повідомлень програміст пише вже в програмі.

2.4.5. Редактор баз даних

Для прикладних програм робота з базами даних (БД) є обов’язковою умовою. Якщо ви пишете програму для автоматизації бізнесу, то вам будуть потрібні робота з базами даних: Працівників, Товарів, Покупців, Рахунків та інші.

Бази даних – це основа автоматизації будь-якої компанії. Тому програмісту в тій чи іншій формі потрібно буде взаємодіяти з базами даних. У цьому йому допомагають редактори БД, які дозволяють керувати інформацією

в базах даних.

Найпотужнішою і зручною системою керування базами даних (СКБД) є **Microsoft Access**, яка входить до складу **Microsoft Office**. Можливості **Access** дуже великі. Ця СУБД дозволяє розробити автоматизацію невеликої компанії, але отриманий продукт не дуже зручно тиражувати через особливості ліцензування **Microsoft Office**.

Найпоширенішими редакторами БД є:

- **PhpMyAdmin**;
- **HeidiSQL**;
- **DBTools Manager**.

2.4.6. Інструмент тестування ПЗ

Як тільки програміст написав програму, йому потрібно переконатися в тому, що вона працює. Для цього існує окремий процес, який називається тестуванням ПЗ.

Суть тестування полягає в тому, що тестувальник виконує пакет тестів і перевіряє відповідність реальної поведінки програми з заданим.

Проблема тестування полягає в тому, що передбачити заздалегідь всі можливі варіанти використання програми неможливо. Тому тестування ПЗ – це більше мистецтво, ніж наука.

Поки програми були не дуже складними, використовувалося так зване «вичерпне тестування», тобто перевірялася робота програми на всіх можливих гілках виконання. Але дуже скоро кількість комбінацій гілок стало перевищувати можливості тестувальників, і зараз проводиться вибіркове тестування. Вибирається деякий основний варіант використання програми і для нього пишуться тести.

Важко знайти універсальні інструменти автоматичного тестування. Як правило, програміст шукає інструмент тестування під конкретну задачу. Якщо ж такого інструменту не знаходиться, то доводиться його писати самому.

2.4.7. Фреймворки

Багато програм мають спільні модулі. Звідси приходить бажання не писати кожен раз програму з нуля, а використовувати який-небудь шаблон. Так з'явилося поняття фреймворку, що в перекладі й означає «каркас».

Фреймворк відрізняється від бібліотеки тим, що бібліотека ніяк не впливає на роботу програміста. Йому досить підключити бібліотеку і він може довільно викликати функції бібліотеки.

Фреймворк диктує принцип побудови програми. Зазвичай при роботі з фреймворком відразу генерується деякий прототип програми, а програміст повинен розвивати цей прототип до готової програми.

Оскільки принципи розробки єдині, то на фреймворку можна досить швидко побудувати цілком робочу програму. Але є й серйозний недолік у використанні фреймворка.

Якщо функціонал програми сильно відрізняється від тих принципів, які використовує фреймворк, то програмісту доводиться постійно винаходити різні прийоми, щоб узгодити вимоги замовника та обмеження фреймворка. Іноді буває так, що використання фреймворка не тільки прискорює, а й, навпаки, уповільнює програмування.

Контрольні запитання та завдання

1. Що таке програмування?

2. Дайте визначення парадигми програмування.

3. Які парадігми прогрмування характерні для імперативного програмування?

4. Перерахвати осеновні риси імперативних мов.

5. Поясніть призначення декларативного програмування.

6. Наведіть основні характеристики процедурного програмування.

7. Що таке архітектура фон Неймана.

8. Для чото призначена машина Тюрінга иа з чого вона складається?

9. Які мови програмвання підтримують парадігму пбстракттної обчислювальної машиги?

10. У чому полягає методологія структурного програмування?

11. Перерахуйте основні поняття аспектно-орієнтоване програмування.

12. У чому полягає ідея об'єктно-орієнтоване програмування?

13. Що передбачає функціональне програмування?

14. На чому засноване логічне програмування?

15. Якими мовами програмування підтримується узазагальнене програмування?

16. Наведіть інструменти для ефективної роботи програміста.

17. З чого складається інтегроване середовище розробки?

18. У чому полягає суть тестування?

19. У чому полягає суть тестування?

20. Чим відрізняється фреймворк від бібліотеки?

3. ОСНОВИ ПРОГРАМУВАННЯ НА МОВІ *C*

3.1. Загальне знайомство з мовою *C*

3.3.1. Походження мови *C*

Мова програмування С була розроблена в лабораторіях **Bell Labs** в період з 1969 по 1973 роки. За словами Деніса Рітчі, найактивніший період творчості припав на 1972 рік. Мову назвали «Сі» (**C** – третя буква латинського алфавіту), тому що багато її особливостей беруть початок від попередньої мови «Бі» (**B** – друга буква латинського алфавіту).

На початок 1973 року мова **C** стає досить сильною, і велика частина ядра ОС **UNIX**, що спочатку написана на асемблері, була переписана на **C**. Це було одне з найперших ядер ОС, яке було написане мовою, відмінною від асемблера.

3.1.2. Стандарти мови С

У 1978 році Брайан Керніган і Денніс Рітчі опублікували першу редакцію книги «Мова програмування Сі». Ця книга, відома серед програмістів як «**K&R**», служила багато років неформальною специфікацією мови.

K&R ввів такі особливості мови: - структури (тип даних **struct**);

- довге ціле (тип даних **long int**);

- ціле без знака (тип даних **unsigned int**); - оператор **+=** і подібні до нього.

Після публікації **K&R C** до мови було додано кілька можливостей:

- функції, що не повертають значення (з типом **void**), і вказівники, що не мають типу (з типом **void ***);

- функції, які повертають об'єднання і структури;

- імена полів даних структур в різних просторах імен для кожної структури; - присвоєння структур;

- специфікатор констант (**const**);

- стандартна бібліотека, яка реалізує більшу частину функцій, введених різними виробниками;

- перелічувальний тип (**enum**);

- дробове число одинарної точності (**float**).

У 1983 році Американський національний інститут стандартів (**ANSI**) сформував комітет для розробки стандартної специфікації **C**. Після закінчення цього процесу, в 1989 році він був затверджений як «Мова програмування Сі» **ANSI X3.159-1989**. Цю версію мови прийнято називати **ANSI C** або **C89**. У 1990 році стандарт **ANSI C** був прийнятий з невеликими змінами Міжнародною організацією зі стандартизації (**ISO**) як **ISO/IEC 9899:1990**.

Однією з цілей цього стандарту була розробка надмножини **K&R C**, що включає багато особливостей мови, які були створені пізніше. Однак комітет зі стандартизації також включив в нього і кілька нових можливостей, таких, як прототипи функцій (запозичені з **C++**) і більш складний препроцесор.

ANSI C зараз підтримують майже всі існуючі компілятори. Будь-яка програма, написана тільки на стандартному **C**, гарантовано буде правильно виконуватися на будь-якій платформі, що має відповідну реалізацію **C**.

Нові особливості *C*99:

- вбудовані функції (**inline**);

- оголошення локальних змінних в будь-якому операторі програми;

- нові типи даних, такі, як **long long int**, булевий тип даних **bool** і тип **complex** для подання комплексних чисел;

- масиви змінної довжини;

- підтримка обмежених вказівників (**restrict**); - іменована ініціалізація структур;

- підтримка однорядкових коментарів, що починаються з **//**; - нові бібліотечні функції, такі, як **snprintf**;

- нові заголовки, такі, як **stdint.h**.

Основні зміни введені в стпгдарті *C*11:

- підтримка багатопоточності;

- поліпшена підтримка

Юнікоду; - узагальнені макроси;

- анонімні структури й об’єднання;

- керування вирівнюванням

об’єктів; - статичні твердження;

- видалення небезпечної функції **gets** (на користь безпечної **gets_s**); - функція **quick_exit**;

- специфікатор функції **_Noreturn**;

- новий режим ексклюзивного відкриття файлу.

3.1.3. Загальна інформація про С

Мова програмування **C** відрізняється мінімалізмом. Автори мови хотіли, щоб програми на ній легко компілювалися за допомогою однопрохідного компілятора, щоб кожній елементарній складовій програми після компіляції відповідало досить невелике число машинних команд, а використання базових елементів мови не задіяло бібліотеку часу виконання. Однопрохідний компілятор компілює програму, не повертаючись назад до вже обробленого тексту. Тому використанню функцій і змінних має передувати їх оголошення. Код на **C** можна легко писати на низькому рівні абстракції, майже як на асемблері. C часто називають мовою середнього рівня або навіть низького рівня, з огляду на те, як близько вона працює до реальних пристроїв. Однак, у суворій класифікації, вона є мовою високого рівня.

C (як і ОС **UNIX**, з якої вона пов’язана) створювалася програмістами і для програмістів, коло яких було б не набагато ширшим за коло розробників мови. Незважаючи на це, сфера використання мови є значно ширшою ніж завдання системного програмування.

С створювалася з однією важливою метою: зробити простішим написання

великих програм з мінімумом помилок за правилами процедурного програмування, не додаючи до підсумкового коду програм зайвих накладних витрат для компілятора, як це завжди роблять мови дуже високого рівня. З цього боку Спропонує наступні важливі особливості:

- просту мовну базу, з якої винесені до бібліотек багато суттєвих можливостей, на кшталт математичних функцій або функцій керування файлами;

- орієнтацію на процедурне програмування, що забезпечує зручність застосування структурного стилю програмування;

- систему типів, що не дозволяє безглуздих операцій;

- використання препроцесора для, наприклад, визначення макросів і включення файлів з початковим кодом;

- безпосередній доступ до пам'яті комп'ютера через використання вказівників;

- мінімальне число ключових слів;

- передачу параметрів до функції за значенням, а не за посиланням (при цьому передача за посиланням емулюється за допомогою вказівників);

- вказівники на функції та статичні змінні; - області дії імен;

- структури й об'єднання – визначені користувачем збірні типи даних, якими можна маніпулювати як одним цілим.

У той же час в **C** відсутні:

- вкладені функції;

- співпрограми; а також засоби:

- автоматичного керування пам'яттю;

- об'єктно-орієнтованого програмування;

- функціонального програмування.

Після своєї появи мова **C** була добре сприйнята, тому що вона дозволяла швидко створювати компілятори для нових платформ, а також дозволяла програмістам досить точно уявляти, як виконуються їхні програми. Завдяки цьому програми, що написані на **C**, є більш ефективними ніж ті, що написані на багатьох інших мовах.

Одним з наслідків високої ефективності та портування **C** стало те, що багато компіляторів, інтерпретаторів і бібліотек інших мов високого рівня часто написані на мові **C**.

3.1.4. Переваги мови С

С – сучасна мова. Вона містить у собі ті керуючі конструкції, які рекомендуються теоретичним і практичним програмуванням. Її структура спонукає програміста використовувати в своїй роботі спадне проектування, структурне програмування та покрокову розробку модулів. Результатом такого підходу є надійна програма, яка добре читається.

С ефективна мова. Її структура дозволяє найкращим чином використовувати можливості сучасних ЕОМ. Написані на мові **C** програми зазвичай відрізняються компактністю та швидкістю виконання.

С – портована (або мобільна) мова. Це означає, що програма, яка написана на С для однієї обчислювальної системи, може бути перенесена з невеликими змінами або взагалі без них, на іншу.

С – потужна та гнучка мова. Наприклад, велика частина потужної та гнучкої ОС **UNIX** написана на мові *С*. Йдеться про компілятори та інтерпретатори інших мов, таких, як Фортран, АПЛ, Паскаль, Лісп, Лого та Бейсік. Крім того, програми, написані на **C**, використовуються для вирішення фізичних і технічних проблем, комп’ютерної графіки і навіть виробництва мультиплікаційних фільмів.

С має низку конструкцій керування, що зазвичай асоціюються з асемблером.

С – зручна мова. Вона досить структурована, щоб підтримувати добрий стиль програмування, і разом з тим не зв’язує обмеженнями.

Мова *С* швидко стає однією з найбільш важливих і популярних мов програмування. Її використання все більш розширюється, оскільки часто програмісти надають перевагу саме мові С після першого знайомства з нею.

3.1.5. Застосування мови *C*, які мають високі вимоги до продуктивності

Деякі застосування мови *C*, які мають високі вимоги до продуктивності, наведені в табл. 3.1.

Таблиця 3.1 – Застосування мови **C**, які мають високі вимоги до продуктивності

Додаток	Опис
1	2
Операційні системи	Мобільність та продуктивність мови **C** є одними з найбільш затребуваних її характеристик при реалізації операційних систем, таких як **Linux** і **Microsoft Windows**, а також **Google Android**. Операційна система **OS X**, що випускається компанією **Apple**, написана на мові **Objective-C**, яка корінням сягає до мови **C**. Ядра ОС в основному написані на **C** і мові асемблера. Решта (особливо графічні інтерфейси) на **C++** (**Linux, Windows**) або **Objective-C** (**Mac OS**). Незначна частина написана на інших мовах: **Python, Perl, Java, Bash, Lisp, Haskell** та ін.
Вбудовані системи	Крім комп'ютерів загального призначення, щороку випускається величезна кількість мікропроцесорів для вбудованих пристроїв. До числа вбудованих систем відносяться системи для навігаційних пристроїв, комплексів «Розумний будинок», охоронних пристроїв, смартфонів, роботів, комплексів керування дорожнім рухом та багато інших. Мова **C** є однією з найпопулярніших мов розробки подібних систем, від яких зазвичай потрібні висока швидкість виконання і низьке споживання пам'яті. Наприклад, автомобільна антиблокувальна система повинна негайно реагувати на уповільнення швидкості обертання коліс, щоб запобігти їх блокуванню; ігрові приставки повинні працювати дуже швидко, щоб забезпечити плавність мультиплікації та швидку реакцію на дії гравця.
Системи реального часу	Системи реального часу часто використовуються для виконання «критично важливих» додатків, що мають дуже жорсткі вимоги до часу відгуку. Наприклад, система керування рухом повітряних суден повинна постійно відстежувати стан і швидкість руху літаків і повідомляти цю інформацію авіадиспетчерам без будь-яких затримок, щоб вони могли вчасно повідомити екіпажу літака про необхідність змінити курс для запобігання зіткнення.
Системи зв'язку	Системи зв'язку повинні швидко керувати рухом величезних обсягів інформації, щоб забезпечити своєчасну доставку аудіо-і відеопотоків адресатам

3.1.6. Майбутнє мови *C*

Багато фірм, що виробляють програмне забезпечення, все частіше звертаються до `C` як до зручної мови для реалізації своїх проектів, оскільки відомо, що `C` дозволяє отримати компактні й ефективні програми. І ці програми можуть бути легко модифіковані та адаптовані до нових моделей ЕОМ!

Мови програмування `C++`, `Java`, `C#`, `UML` та інші мають «сішну» семантику. `C` використовується фірмами, які розробляють програмне забезпечення, студентами, які навчаються програмуванню. І якщо ви хочете працювати в сфері, яка пов'язана з програмуванням, то одне з перших питань, на яке ви повинні будете відповідати «Так» – це питання «Чи вмієте ви програмувати на `C`?».

3.1.7. Базові поняття мови *C*

Починаючи вивчати нову алгоритмічну мову програмування, необхідно з'ясувати наступні питання:

1) Який алфавіт мови і як правильно записувати її лексеми?

2) Які типи даних прийняті в мові і як вони визначаються (описуються)?

3) Які операції над даними допустимі в мові, як будуються з їх допомогою вирази і як вони виконуються?

4) Яка структура програми, в якій послідовності розміщуються оператори, опис та визначення?

5) Як виводити (представляти користувачеві) результати роботи програми?

6) Як реалізовані оператор присвоювання, умовні оператори й оператори переходу?

7) Як вводити вхідні дані для програми?

8) Які спеціальні конструкції для організації циклів є в мові?

9) Який апарат підпрограм (процедур) та (або) підпрограм-функцій?

Потім слід приступати до складання програм, поглиблюючи в ході програмування знання мови.

В ***алфавіт*** *мови C* входять:

- великі та малі літери латинського алфавіту (**A, B, ..., Z, a, b, ... , z**);

- цифри: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**;

- спеціальні знаки:

```
"   { }   |   [ ]   ( )
+   -     /   %     \
;   '     .   :     ?
<   =     >   _     !
&   *     #   ~     ^
```

- символи, що не відображаються («узагальнені пробільні символи»), які використовуються для відділення лексем одна від одної (наприклад, пробіл, табуляція, перехід на новий рядок).

Коментар формується як послідовність знаків (символів), обмежена зліва знаками **/***, а праворуч – знаками ***/**, наприклад:

```
/* Це коментар */
```

У стандартній мові С коментарі заборонено вкладати один в одного.

У коментарях, рядках і символьних константах можуть використовуватися й інші літери (наприклад, українські літери).

У стандарті **C99** додана підтримка однорядкових коментарів, що починаються з **//**.

```
// Це коментар
```

Лексема – це одиниця тексту програми, яка при компіляції сприймається як єдине ціле й за змістом не може бути розділена на більш дрібні елементи.

У мові **C** існує шість класів лексем:

- ідентифікатори;

- службові (ключові) слова; - константи;

- рядки (рядкові константи); - операції (знаки операцій);

- роздільники (знаки пунктуації).

Ідентифікатор – це послідовність букв, цифр і символів підкреслення

'_', що починається з букви або символу підкреслення.

Приклади ідентифікаторів:

```
KOM_16, Size88, _MIN, TIME, time
```

Великі та малі літери розрізняються, тобто два останніх ідентифікатора є різними.

Ідентифікатори можуть мати будь-яку довжину, але компілятор враховує не більше 31-го символу від початку ідентифікатора. У деяких компіляторах це обмеження ще більш жорстке, і враховуються тільки перші 8 символів будь-якого ідентифікатора. В цьому випадку ідентифікатори **NUMBEROFROOM** і **NUMBEROFTEST** в програмі будуть не розрізнені.

Ідентифікатори, які зарезервовані в мові, тобто такі, які не можна використовувати в якості імен, що вільно обираються програмістом, називають **службовими словами**.

Службові слова визначають типи даних, класи пам'яті, кваліфікатори типу, модифікатори, псевдозмінні (**__FILE__**) та оператори.

У стандартах мови **C** визначені такі базові службові слова:

- **C89 (ANSI C):**

auto	**break**	**case**	**char**
const	**continue**	**default**	**do**
double	**else**	**enum**	**extern**
float	**for**	**goto**	**if**
int	**long**	**register**	**return**
short	**signed**	**sizeof**	**static**
struct	**switch**	**typedef**	**union**
unsigned	**void**	**volatile**	**while**

- **C99:**

___Bool _Complex _Imaginary _inline restrict

C11:

_Alignas **_Alignof** **_Atomic**

_Generic **_Noreturn** **_Static_assert**

_Thread_local

За смисловим навантаженням службові слова групуються наступним чином. Для позначення типів даних використовуються специфікатори та кваліфікатори типів.

До ***специфікаторів типів*** відносяться:

- **char** – символьний;

- **double** – дійсний подвійної точності з рухомою крапкою;

- **enum** – перелічуваний (перерахування) визначення цілочислових констант, для кожної з яких вводяться ім’я і значення;

- **float** – дійсний з рухомою крапкою; - **int** – цілий;

- **long** – цілий збільшеної довжини (довге ціле);

- **short** – цілий зменшеної довжини (коротке ціле);

- **struct** – структура (структурний тип);

- **signed** – знаковий, тобто ціле зі знаком (старший біт вважається знаковим);

- **union** – об’єднання (об’єднуючий тип);

- **unsigned** – беззнаковий, тобто ціле без знака (старший біт не вважається знаковим);

- **void** – відсутність значення;

- **typedef** – вводить синонім позначення типу (визначає скорочене найменування для позначення типу).

До ***кваліфікаторів типу*** належать:

- **const** – кваліфікатор об’єкта, що має постійне значення, тобто доступного тільки для читання;

- **volatile** – кваліфікатор об’єкта, значення якого може змінитися без явних вказівок програміста.

Кваліфікатори типу інформують компілятор про необхідність та (або) можливості особливої обробки об'єктів в процесі оптимізації коду програми.

Для позначення ***класів пам'яті*** використовуються:

- **auto** – автоматичний;
- **extern** – зовнішній;
- **register** – регістровий;
- **static** – статичний.

Для побудови операторів використовуються ***службові слова***:

- **break** – вийти з циклу або перемикача;
- **continue** – завершити поточну ітерацію циклу (продовжити цикл, перейшовши до наступної ітерації);
- **do** – виконувати (заголовок оператора циклу з постумовою);
- **for** – для (заголовок оператора параметричного циклу);
- **goto** – перейти (безумовний перехід);
- **if** – якщо (позначення умовного оператора);
- **return** – повернення (з функції);
- **switch** – перемикач;
- **while** – поки (заголовок циклу з передумовою або завершення циклу **do**).

До службових слів також віднесені наступні ***ідентифікатори***:

- **default** – визначає дії при відсутності потрібного варіанту в операторі **switch**;
- **case** – визначає варіант в операторі **switch**;
- **else** – входить до оператору **if**, визначаючи альтернативну гілку;
- **sizeof** – операція визначення розміру операнда (в байтах).

До числа службових слів, визначених стандартом мови, модифікатори не входять, однак їх необхідно знати, працюючи з конкретною реалізацією мови. Практично у всіх шістнадцяткових реалізаціях компіляторів для

IBM PC використовуються наступні модифікатори:

asm	**cdecl**	**_cs**	**_ds**	**_es**	**far**
fortran	**huge**	**interrupt**	**near**	**pascal**	**_ss**

Крім основних (визначених стандартом мови) службових слів і модифікаторів, конкретні реалізації мови **C** (конкретні компілятори) містять власні розширення списку службових (ключових) слів. Наприклад, компілятор **Borland C++** для мови **C** додатково вводить такі модифікатори (перераховані не всі):

_cdecl	**_export**	**_far**
_fastcall	**_loadds**	**_near**
_pascal	**_saveregs**	**_seg**

Для роботи з регістрами в тому ж компіляторі ***Borland C++*** визначено набір псевдозмінних, доступних з програм на мові *C* і заборонених для використання в інших цілях:

_AH	**_AL**	**_AX**	**_BH**	**_BL**	**_BP**	**_BX**
_CH	**_CL**	**_CS**	**_CX**	**_DH**	**_DI**	**_DL**
_DS	**_DX**	**_ES**	**_FLAGS**	**_SI**	**_SP**	**_SS**

Конструкції мови, в яких використовуються службові слова, будемо визначати за необхідністю. Можна було б не перераховувати всі службові слова, а вводити їх у міру викладення мови, проте їх заборонено використовувати як імена, що обираються програмістом, і тому для уникнення можливих помилок список службових слів потрібен вже на даному етапі.

Слід додати ще одну угоду, якої зазвичай дотримуються автори компіляторів і стандартних бібліотек мови **C**. Ідентифікатори, що починаються з одного або двох символів підкреслення «_», зарезервовані для використання в бібліотеках і компіляторах. Тому такі ідентифікатори не рекомендується вибирати в якості імен в прикладній програмі на мові **C**. Наступна угода

щодо імен відноситься вже не до стандарту й не до реалізацій, а відображає стиль оформлення тексту програми. Рекомендується при програмуванні імена констант записувати ПОВНІСТЮ ВЕЛИКИМИ ЛІТЕРАМИ.

3.2. Базові поняття мови *C*

3.2.1. Константи

За визначенням, константа – це значення, яке не може бути змінено. Синтаксис мови визначає п’ять типів констант:

- символи;

- константи перелічуваного типу; - дійсні числа;

- цілі числа;

- нульовий вказівник («**`null`**-вказівник»).

Стандарт ввів додатковий тип констант «символ розширеної форми представлення», але зараз його підтримує незначна кількість компіляторів.

Усі константи, крім нульового вказівника, віднесені в мові **`C`** до арифметичних констант.

Символи (символьні константи). Для зображення окремих знаків, що мають індивідуальні внутрішні коди, використовуються символьні константи. Кожна символьна константа – це лексема, яка складається з зображення символу та апострофів, які її обмежують. Наприклад: **`'A', 'a', 'B', '8', '0', '+', ';'`** та інші.

Усередині апострофів можна записати будь-який символ, що зображується на дисплеї або принтері в текстовому режимі. Однак в ЕОМ використовуються також коди, які не мають графічного представлення на екрані дисплея, клавіатури або принтері. Прикладами таких кодів є код переходу курсору дисплея на новий рядок або код повернення каретки (повернення курсору на початок поточного рядка). Для зображення у програмі відповідних символьних констант використовуються

комбінації з декількох символів, що мають графічне представлення. Кожна така комбінація починається з символу **'\'** (зворотна коса риска – **backslash**). Такі набори літер, що починаються з символу **'\'**, в літературі з мови **C** називають ***керувальними послідовностями***.

Нижче наводиться їх перелік:

'\n' – новий рядок;

'\t' – горизонтальна табуляція;

'\r' – повернення каретки (курсору) до початку рядка; **'\\'** – зворотна коса риска ****;

'\'' – апостроф (символ «одинарні лапки»); **'\"'** – лапки (символ «подвійні лапки»); **'\0'** – нульовий символ;

'\a' – сигнал-дзвоник;

'\b' – повернення на одну позицію (на один символ); **'\f'** – переведення (прогін) сторінки;

'\v' – вертикальна табуляція; **'\?'** – знак питання.

Зверніть увагу на те, що розглянуті константи зображуються двома і більше літерами, а позначають вони одну символьну константу, що має індивідуальний двійковий код. Керувальні послідовності є окремим випадком ескейп-послідовностей (**ESC-sequence**), до яких також належать лексеми виду **'\ddd'** або **'\xhh'**, або **'\Xhh'**.

'\ddd' – вісімкове представлення будь-якої символьної константи, де буква **d** – це вісімкова цифра (від **0** до **7**). Наприклад, **'\017'** або **'\233'**.

'\xhh' або **'\Xhh'** – шістнадцяткове представлення будь-якої символьної константи, де **h** – це шістнадцяткова цифра (від **0** до **F**). Наприклад, **'\x0b'**, **'\x1A'** та інші.

Символьна константа (символ) має цілий тип, тобто символи можна використовувати в якості цілочислових операндів у виразах.

Синтаксисом мови визначені десяткові, шістнадцяткові та вісімкові ***цілі константи***. Основа системи числення визначається префіксом в запису

константи. Для десяткових констант префікс не використовується. Десяткові цілі визначені як послідовності десяткових цифр, що починаються не з нуля (якщо це не число нуль): **44, 684, 0, 1024**.

Послідовність цифр, що починається з **0** та не містить десяткових цифр старше **7**, сприймається як вісімкова константа:

016 – вісімкове представлення десяткового цілого **14**.

Послідовність шістнадцяткових цифр (**0, 1, ..., 9, A, B, C, D, E, F**), перед якою записані символи **0x** або **0X**, вважається шістнадцятковою константою:

0x16 – шістнадцяткове представлення десяткового цілого **22**; **0XFF** – шістнадцяткове представлення десяткового цілого **255**.

Кожна конкретна реалізація мови вводить свої обмеження на граничні значення констант.

Для подання дійсних (не цілих) чисел використовуються ***дійсні константи***, які наведені в пам'яті ЕОМ у формі з рухомою крапкою.

Кожна дійсна константа складається з наступних частин: - ціла частина (десяткова ціла константа);

- десяткова крапка;

- дробова частина (десяткова ціла константа); - ознака показника «**e**» або «**E**»;

- показник десяткової степені (десяткова ціла константа, можливо, зі знаком).

Під час запису констант з рухомою крапкою можуть опускатися ціла або дробова частина (але не одночасно); десяткова крапка або символ експоненти з показником степені (але не одночасно).

Приклади констант з рухомою крапкою:

44.   3.14159   44e0   .314159E1   0.0

Машинне подання (код) програми на мові **C** передбачає, що кожна константа, яка введена в програмі, займає в ЕОМ деяку комірку пам'яті.

Розміри цієї комірки пам’яті та інтерпретація її вмісту визначаються типом відповідної константи.

Майже всі компілятори відводять символьним константам (символам) по одному байту (вісім біт). Тим самим вводиться обмеження на всю різноманітність символьних констант – їх внутрішні коди повинні знаходитися в діапазоні від **0** до **255**. У мовах багатьох країн світу не весь набір символів (букв і знаків) може бути представлений за допомогою одного байта.

Для цілих і дійсних констант кожна реалізація компілятора з мови **C** може визначати свої обмеження. У табл. 3.2 наведені межі, виходячи з яких компілятори, які реалізовані на IBM-сумісних ПЕОМ, обирають типи цілих констант. Наприклад, всі цілі константи в діапазоні від **0** до **32767** мають тип **int**, тобто будуть представлені в пам’яті 2 байтами (16 бітами).

Таблиця 3.2 – Цілі константи та типи даних, які для них обираються

Діапазони значень констант			Тип даних
десяткові	вісімкові	шістнадцяткові	
від 0 до 32767	від 00 до 077777	від 0x0000 до 0x7FFF	**int**
–	від 0100000 до 0177777	від 0x8000 до 0xFFFF	**unsigned int**
від 32768 до 2147483647	від 020000 до 017777777777	від 0x10000 до 0x7FFFFFFF	**long**
від 2147483648 до 4294967295	від 020000000000 до 037777777777	від 0x80000000 до 0xFFFFFFFF	**unsigned long**
> 4294967295	> 037777777777	> 0xFFFFFFFF	**unsigned long**

В табл. 3.3 наведена інформація про дані дійсних типів.

Таблиця3.3 – Дані дійсних типів

Тип даних	Розмір, біт	Діапазон абсолютних величин
float	32	від 3.4E-38 до 3.4E+38
double	64	від 1.7E-308 до 1.7E+308
long double	80	від 3.4E-4932 до 1.1E+4932

Дійсна константа **3.141592653589793** буде сприйнята як така, що має тип **double**, і їй буде виділено **8** байт (**64** біта). Такий саме тип обирається для константи **3.14**, оскільки за замовчуванням всім дійсним константам присвоюється тип **double**.

Якщо програміста не влаштовує тип, який компілятор приписує константі, то тип можна явно вказати в запису константи за допомогою суфіксів:

F (або **f**) – **float** (для дійсних);

U (або **u**) – **unsigned** (для цілих);

L (або **l**) – **long** (для цілих і дійсних).

Наприклад:

3.14159F – константа типу **float** (виділяється **4** байта);

3.14L – константа типу **long double** (виділяється **10** байт).

За допомогою суфікса **U** (або **u**) можна подати цілу константу у вигляді беззнакового цілого.

Наприклад:

50000U – константа типу **unsigned int**.

Константі **50000U** виділяються **2** байта замість чотирьох, як було б при відсутності суфікса (табл. 3.2). В цьому випадку, тобто для **unsigned int** знаковий біт використовується для подання одного з розрядів коду числа і діапазон значень стає від **0** до **65535**.

Суфікс **L** (або **l**) дозволяє виділити для цілої константі **4** байта (**32** біта): **500L** – константа типу **long**, для якої виділяється **4** байта;

0L – ціла константа типу **long** довжиною **4** байта.

Спільне використання в будь-якому порядку суфіксів **U** (або **u**) і **L** (або **l**) дозволяє приписати цілій константі тип **unsigned long**, і вона займе в пам'яті **32** розряди (біта), причому знаковий розряд буде використовуватися для подання розряду коду (а не знака).

Приклади:

0LU – ціла константа типу **unsigned long** довжиною **4** байта;

2424242424UL – константа типу **unsigned long**.

Null-вказівник, який зветься ***нульовим вказівником***, це єдина не арифметична константа. Її роль і функціональні можливості стануть зрозумілими при вивченні вказівників. У конкретних реалізаціях **null**-вказівник може бути представлений або як **0**, або як **0L**, або як іменована константа **NULL**. Тут потрібно відзначити, що значення константи **NULL** не повинно бути нулем, і має право не збігатися з кодом символу «**0**».

Цілочислові іменовані константи можна вводити за допомогою перелічення:

enum тип_перелічення { список_іменованих_констант };

де **enum** – службове слово, що вводить перелічення;

тип_перелічення – його назва – необов'язковий довільний ідентифікатор; список_іменованих_констант – розділена комами послідовність

ідентифікаторів або іменованих констант виду:

ім'я_константи = значення константи

Приклади:

```
enum { ONE = 1, TWO, THREE, FOUR };
            enum  DAY  {  SUNDAY,  MONDAY,
      TUESDAY,    WEDNESDAY,    THURSDAY,
      FRIDAY, SATURDAY };
enum BOOLEAN { NO, YES };
```

Якщо в списку немає жодного елемента зі знаком **'='**, то значення констант починаються з **0** і збільшуються на **1** зліва направо. Таким чином, **NO** дорівнює **0**, **YES** дорівнює **1**, **SUNDAY** має значення **0** і **FRIDAY** має значення **5**. Іменована константа зі знаком **'='** отримує відповідне значення (**ONE = 1**), а наступні за нею іменовані константи без явних значень збільшуються на **1** кожна. У нашому прикладі **TWO** дорівнює **2**, **THREE** дорівнює **3**, **FOUR** дорівнює **4**.

3.2.2. Рядки (рядкові константи)

Формально рядки (відповідно до стандарту) не належать до констант мови

C, а являють собою окремий тип його лексем. Для них в літературі використовується ще одна назва «рядкові літерали». Рядкова константа визначається як послідовність символів, яка поміщається у подвійні лапки (не в апострофи):

```
"Зразок рядка"
```

Серед символів рядка можуть бути ескейп-послідовності, тобто поєднання знаків, що відповідають символам, які не відображаються, або символам, що задаються їх внутрішніми кодами. У цьому випадку, як і в представленнях окремих символьних констант, їх зображення починаються зі зворотної косої риски «****»:

```
"\n Текст \n розміститься \n в 3-х рядках дисплея"
```

Представлення рядкових констант у пам'яті ЕОМ відповідають таким правилам. Всі символи рядка розміщуються один за одним, і кожен символ (в тому числі й той, що поданий у вигляді ескейп-послідовності) займає рівно **1** байт. В кінці запису рядкової константи компілятор поміщає символ **'\0'**.

Таким чином, кількість байтів, що виділяється в пам'яті ЕОМ для представлення значення рядка, рівно на **1** більше, ніж число символів у записі цієї рядкової константи:

```
"Цей рядок займає в пам\'яті ЕОМ 40
байт." "Рядок у 17 байт."
```

При роботі з символьною інформацією потрібно пам'ятати, що довжина константи **'F'** дорівнює **1** байту, а довжина рядка **"F"** дорівнює **2** байтам.

Під час запису рядкових констант можливе розміщення однієї константи

в декількох рядках текстового файлу з програмою. Для цього використовується наступне правило.

Якщо в послідовності символів (літер) константи зустрічається літера **'\'**, за якою до ознаки **'\n'** кінця рядка текстового файлу розміщені тільки пробіли, то ці пробіли разом з символом **'\'** і закінченням **'\n'** видаляються, і продовженням рядкової константи вважається наступний рядок тексту. Наприклад, наступний текст являє собою одну рядкову константу:

```
    "Кафедра␣мультимедійних␣\
␣␣␣інформаційних␣технологій␣і␣систем."
```

У програмі ця константа буде еквівалентна такій константі:

```
    "Кафедра␣мультимедійних␣␣␣␣інформаційних␣технологій␣і␣систем."
```

Початкові (ліві) пробіли в продовженні константи на новому рядку не видаляються, а вважаються такими, що входять до складу рядкової константи.

Дві рядкові константи, між якими немає інших роздільників, крім узагальнених символів пробілів (пробіл, табуляція, кінець рядка та ін.); сприймаються як одна рядкова константа. Таким чином,

```
    "Кафедра␣мультимедійних"␣"␣інформаційних␣технологій␣і␣систем."
```

сприймається як одна константа:

```
    "Кафедра␣мультимедійних␣інформаційних␣технологій␣і␣систем."
```

Таким самим правилам відповідають і рядкові константи, які розміщені на різних рядках. Як один рядок буде сприйнята послідовність

```
    "Кафедра"
    ␣␣␣␣"␣мультимедійних"
    ␣␣␣␣␣␣␣␣"␣інформаційних"
    ␣␣␣␣␣␣␣␣␣␣␣␣"␣технологій і систем."
```

Ці чотири рядкові константи еквівалентні одній:

```
"Кафедра␣мультимедійних␣інформаційних␣технологій␣і␣систем."
```

Зверніть увагу, що в результуючий рядок тут не включаються початкові

пробіли перед кожною константою-продовженням.

3.2.3. Змінні та іменовані константи

Одним з основних понять мови `C` є об'єкт – іменована область пам'яті. Окремим випадком об'єкта є **змінна**. Особливість змінної полягає в можливості пов'язувати з її ім'ям різні значення, сукупність яких визначається типом змінної. При задаванні значення змінної у відповідну їй область пам'яті поміщається код цього значення. Доступ до значення змінної забезпечує її ім'я, а доступ до комірки пам'яті можливий тільки за її адресою. Про взаємозв'язки імен та адрес буде детально говоритися в лекції, що присвячена вказівникам і роботі з пам'яттю ЕОМ. Сьогодні буде цілком достатньо інтерпретувати поняття змінної як пару «ім'я – значення».

Кожна змінна перед її використанням у програмі повинна бути визначена, тобто для змінної повинна бути виділена пам'ять. Розмір комірки пам'яті, що виділяється для змінної, і інтерпретація вмісту можуть відрізнятися залежно від типу, зазначеного у визначенні змінної.

Відповідно до типів значень, допустимих в мові `C`, розглянемо символьні, цілі і дійсні змінні автоматичної пам'яті. Про класи пам'яті (одним з яких є клас автоматичної пам'яті) будемо докладно говорити пізніше. Зараз досить ввести тільки змінні автоматичної пам'яті, які існують в тому блоці, де вони визначені. У найбільш поширеному випадку таким блоком є текст головної функції програми (функції **`main`**).

Найпростіша форма визначення змінних:

```
тип список_імен_змінних;
```

де `список_імен_змінних` – це обрані програмістом ідентифікатори, які в списку розділяються комами; **`тип`** – один зі згадуваних вже типів (у зв'язку з константами).

У версії **`C89`** визначено такі основні типи даних:

char – символьний; **int** – цілочисловий;

float – з рухомою крапкою;

double – з рухомою крапкою подвійної точності; **void** – порожній.

До перелічених вище типів у версії **C99** додані наступні:

_Bool– логічний або булевий (так/ні);

_Complex – комплексний;

_Imaginary– уявний;

Один з нових типів даних, що з'явилися в **C99**, – це **_Bool**, в якому можна зберігати значення **1** і **0** (**true** і **false**). **_Bool** це цілий тип даних. У **C99** є заголовок **<stdbool.h>**, в якому визначені імена макросів **bool**, **true** і **false**. Таким чином, можна легко створювати код, сумісний з **C/C++.**

Причина того, що в якості ключового слова вказується **_Bool**, а не **bool**, полягає в тому, що в багатьох уже наявних **C**-програмах визначені їх власні варіанти **bool**. Визначаючи логічний тип як **_Bool**, **C99** дає можливість не міняти вже написаний код. Однак в нові програми краще вставляти **<stdbool.h>**, а потім використовувати ім'я макросу **bool**.

Стандарт **C99** з'явився разом з новою для **C** підтримкою арифметичних операцій з комплексними числами. Ця підтримка включає в себе ключові слова **_Complex** і **_Imaginary**, додаткові заголовки та кілька нових бібліотечних функцій. Однак ніяких реалізацій не вимагається, щоб реалізувати типи уявних чисел (**imaginary types**), а автономні програми (які обходяться без операційної системи) не зобов'язані підтримувати комплексні типи. Арифметичні операції з комплексними числами з'явилися в **C99** для спрощення програмування чисельних методів.

Заголовок **<complex.h>** визначає (крім усього іншого) макроси **complex** і **imaginary**, які в результаті макропідстановки перетворюються в **_Complex** і **_Imaginary**. Таким чином, в нові програми краще вставляти **<complex.h>**, а потім використовувати макроси **complex** та

imaginary.

У стандарті **C11** увесь пакет підтримки комплексних чисел є необов’язковим.

У стандарті **C99** з’явилися нові для **C** типи даних **long long int** та **unsigned long long int**. Діапазон значень типу даних **long long int** не вужче, ніж інтервал від $-(2^{63}-1)$ до $(2^{63}-1)$. Діапазон значень типу даних **unsigned long long int** зобов’язаний містити інтервал від **0** до $2^{64}-1$. Типи **long long** дозволяють підтримувати **64**-розрядні цілі значення за допомогою вбудованого типу.

Основні типи даних мови C наведені в табл. 3.4.

Таблиця 3.4 – Основні типи даних мови *C*

Тип даних		Систематичне ім’я	Інше ім’я
Цілий	беззнаковий	_Bool	bool
		unsigned char	
		unsigned short	
		unsigned int	unsigned
		unsigned long	
		unsigned long long	
	[без]знаковий	char	
	знаковий	signed char	
		signed short	shot
		signed int	Signed або int
		signed long	long
		signed long long	long long
З рухомою крапкою	дійсний	float	
		double	
		long double	
	комплексний	float _Complex	float complex
		float _Imaginary	float imaginary
		double _Complex	double complex
		double _Imaginary	double imaginary
		long double _Complex	long double complex
		long double _Imaginary	long double imaginary

Типи з сірим фоном не враховують арифметику, вони підвищують свій

тип, перш ніж здійснювати арифметичні дії. Тип **char** особливий, оскільки він може бути як без знака, так і зі знаком (в залежності від платформи). Всі типи в табл. 3.4 вважаються різними типами, навіть якщо вони мають однаковий клас і точність.

Кожен з цілочисельних типів може бути визначений або як знаковий **signed** або як беззнаковий **unsigned** (за замовчуванням **signed**).

Різниця між цими двома типами – в правилах інтерпретації старшого біта внутрішнього представлення. Специфікатор **signed** вимагає, щоб старший біт внутрішнього представлення сприймався, як знаковий; **unsigned** означає, що старший біт внутрішнього представлення входить до коду відповідного числового значення, яке вважається в цьому випадку беззнаковим.

Вибір знакового або беззнакового представлення визначає граничні значення, які можна використовувати за допомогою описаної змінної. Наприклад, на **IBM PC** змінна типу **unsigned int** дозволяє представити числа від **0** до **65535**, а змінній типу **signed int** (або просто **int**) відповідають значення в діапазоні від **-32768** до **+32767**. Щоб глибше зрозуміти відмінність між цілою величиною та цілої величиною без знака, слід звернути увагу на результат виконання унарної операції «–» (мінус) над цілою величиною та цілої величиною без знака. Для цілої величини результат очевидний і тривіальний. Результатом при використанні цілої величини без знака є 2^n – (значення величини без знака), де **n** – кількість розрядів, відведена для представлення величини без знака.

Наприклад, якщо змінна **k** типу **int** дорівнює **16**, то значенням **-k** буде **-16**. Якщо змінна **b** типу **unsigned int** дорівнює **16**, то значенням **-b** на **IBM PC** є **65520**.

За замовчуванням, при відсутності в якості префікса ключового слова, будь-який цілий тип вважається знаковим (**signed**). Таким чином, вживання спільно зі службовими словами **char, short, int, long** префікса

signed є зайвим. Припустимо окреме використання позначень (специфікаторів) «знаковості». При цьому:

signed еквівалентно **signed int**;

unsigned еквівалентно **unsigned int**.

Приклади визначень цілочислових змінних:

```
char  symbol, cc;
unsigned char code;
int number, row;
unsigned long long_number;
```

Зверніть увагу на необхідність символу «крапка з комою» в кінці кожного визначення.

Стандартом мови введені такі дійсні типи:

float – дійсний тип одинарної точності;

double – дійсний тип подвійної точності;

long double– дійсний тип максимальної точності.

Значення всіх дійсних типів в ЕОМ представляються з «рухомою крапкою», тобто з мантисою і порядком, як було розглянуто при визначенні констант. Приклади визначень дійсних змінних:

```
Float      x, X, CC3, pot_8;
double     a, Stop, B4;
```

Граничні значення констант (і відповідних змінних) розробники компіляторів вправі вибирати самостійно виходячи з апаратних можливостей комп'ютера. Однак при такій свободі вибору стандарт мови вимагає, щоб для значень типу **short**і **int**було відведено не менше **16**біт, для **long**– не менше **32**біт. При цьому розмір **long**повинен бути не менше розміру **int**, а **int**– не менше **short**. Граничні значення арифметичних констант і змінних для більшості компіляторів, реалізованих на **IBM PC**, наведені в табл. 5.4.

Граничні значення дійсних змінних збігаються з граничними значеннями

відповідних констант (табл. 5.2).

Граничні значення цілочисельних змінних збігаються з граничними значеннями відповідних констант (табл. 3.2). Табл. 3.5 містить і граничні значення для тих типів, які не включені до табл. 3.3.

Таблиця 3.5 – Граничні значення арифметичних констант і змінних

Тип даних	Розмір, біт	Діапазон значень
unsigned char	8	0 ... 255
char	8	-128 ... 127
enum	16	-32768 ... 32767
unsigned int	16	0 ... 65535
short int (short)	16	16 -32768 ... 32767
unsigned short	16	0 ... 65535
int	16	-32768 ... 32767
unsigned long	32	0 ... 4294967295
long	32	-2147483648... 2147483648
float	32	3.4E-38... 3.4E+38
double	64	1.7E-308... 1.7E+308
long double	80	80 3.4E-4932 ... 1.1E+4932

Відповідно до синтаксису мови змінні автоматичної пам’яті після визначення за замовчуванням мають невизначені значення. Сподіватися на те, що вони дорівнюють, наприклад, 0, не можна. Однак змінним можна присвоювати початкові значення, явно вказуючи їх у визначеннях:

```
тип ім'я_змінної = початкове_значення;
```

Цей прийом має назву ініціалізація. На відміну від присвоєння, яке здійснюється в процесі виконання програми, ініціалізація виконується при виділенні для змінної комірки пам’яті. Приклади визначень з ініціалізацією:

```
float  pi = 3.1415, cc = 1.23;
unsigned int year = 2019;
```

У мові **C**, крім змінних, можуть бути визначені константи, що мають фіксовані назви (імена). У якості імен констант використовуються довільно

обрані програмістом ідентифікатори, що не збігаються з ключовими словами та з іншими іменами об’єктів. Традиційно прийнято, що для позначень констант обирають ідентифікатори з ВЕЛИКИХ ЛІТЕР ЛАТИНСЬКОГО АЛФАВІТУ І СИМВОЛІВ ПІДКРЕСЛЕННЯ. Така угода дозволяє при перегляді великого тексту програми на мові **C** легко відрізняти імена змінних від назв констант.

Перша можливість визначення іменованих констант – це константи перелічуваного типу, що вводяться з використанням службового слова **enum**.

Другу можливість вводити іменовані константи забезпечують визначення такого виду:

```
const тип імя_константи = значення_константи;
```

Тут **const** – кваліфікатор типу, який вказує, що визначений об’єкт має постійне значення, тобто доступний лише для читання; **тип** – один з типів об’єктів; `ім'я_константи` – ідентифікатор; `значення_константи` має відповідати її типу.

Приклади:

```
const double E = 2.718282;
const long M = 99999999;
const F = 765;
```

В останньому визначенні тип константи не вказано. За замовчуванням йому приписується тип **int**.

Третю можливість вводити іменовані константи забезпечує препроцесорна директива

```
#define імя_константи значення_константи
```

Зверніть увагу на відсутність символу «крапка з комою» в кінці директиви. Докладному розгляду директиви `#define` буде присвячена окрема лекція. Зараз тільки згадується про можливість визначати за її допомогою іменовані константи. Крім того, слід зазначити, що в конкретні реалізації компіляторів

за допомогою директив `#define` включають цілий набір іменованих констант з фіксованими іменами.

Відмінність визначення іменованої константи

```
const double   E = 2.718282;
```

від визначення препроцесорної константи з таким же значенням

```
#define   EULER 2.718282
```

полягає в тому, що у визначенні константи **E** явно задається її тип, а при препроцесорному визначенні константи `EULER` її тип визначається «зовнішнім виглядом» значення константи. Наприклад, таке визначення

```
#define NEXT 'Z'
```

вводить позначення `NEXT` для символьної константи `'Z'`. Це відповідає такому визначенню:

```
const char NEXT = 'Z';
```

Однак відмінності між звичайною іменованою константою і препроцесорною константою, що вводиться директивою `#define`, є набагато глибшими і більш принциповими. До початку компіляції текст програми на мові **C** обробляється спеціальним компонентом транслятора – препроцесором. Якщо в тексті зустрічається директива

```
#define   EULER     2.718282
```

а нижче її в тексті використовується ім'я константи `EULER`, наприклад, в такому вигляді:

```
double mix = EULER;
d = alfa * EULER;
```

то препроцесор замінить кожне позначення `EULER` на її значення і сформує такий текст:

```
double mix =2.718282;
d * = alfa *2.718282;
```

Далі текст від препроцесора надходить до компілятора, який вже «і не згадає» про існування імені EULER, яке було використане у препроцесорній директиві #define. Константи, що визначаються на препроцесорному рівні за допомогою директиви #define, дуже часто використовуються для задавання розмірів масивів, що буде продемонстровано пізніше.

Отже, основна відмінність констант, що визначаються препроцесорними директивами #define, полягає в тому, що ці константи вводяться в текст програми до етапу її компіляції. Спеціальний компонент транслятора – препроцесор обробляє вхідний текст програми, підготовлений програмістом, і робить в цьому тексті заміни та підстановки. Припустимо, що у вхідному тексті зустрічається директива:

```
#define ZERO   0.0
```

Це означає, що кожне наступне використання в тексті програми імені ZERO буде замінюватися на 0.0.

Розглянемо деякі принципи роботи препроцесора. Його основна відмінність від інших компонентів транслятора полягає в тому, що обробка програми виконується тільки на рівні її тексту. На вході препроцесора – текст з препроцесорними директивами, на виході препроцесора – модифікований текст без препроцесорних директив.

Текст до препроцесора:

```
#define   PI   3.141593
#define   ZERO 0.0
...
    if(r > ZERO) ...  // Порівняння з константою ZERO
    D = 2 * PI * r;   // Довжина кола радіуса r
...
```

Текст після препроцесора:

```
...
if(r > 0.0)                    // Порівняння з константою ZERO
```

```
D = 2 * 3.141593 * r; ...// Довжина кола радіуса r
```

Цей вихідний модифікований текст змінений у порівнянні з вхідним текстом за рахунок виконання препроцесорних директив, але самі препроцесорні директиви у вихідному тексті відсутні. Повністю всі препроцесорні директиви будуть розглянуті в окремій лекції. У зв’язку з іменованими константами тут розглядається лише одна з можливостей директиви `#define` – проста підстановка.

Імена `PI` і `ZERO` після роботи препроцесора замінені в тексті програми на певні в двох директивах `#define` значення (`3.141593` і `0.0`).

Зверніть увагу, що підстановка не виконується в коментарях і в рядкових константах. У розглянутому прикладі ідентифікатор `ZERO` залишився без змін в коментарі (`// Порівняння з константою ZERO`).

Саме за допомогою набору іменованих препроцесорних констант стандарт мови **C** рекомендує авторам компіляторів визначати граничні значення всіх основних типів даних. Для цього в мові визначено набір фіксованих імен, кожне з яких є ім’ям однієї з констант, що визначають те чи інше граничне значення. Наприклад:

```
FLT_MAX - максимальне число з плаваючою точкою типу float;
CHAR_BIT - кількість бітів в байті;
INT_MIN- мінімальне значення для даних типу int.
```

Щоб використовувати зазначені іменовані препроцесорні константи, недостатньо записати їх імена у програмі. Попередньо до тексту програми необхідно включити препроцесорну директиву такого виду:

```
#include <ім’я_заголовного_файлу>
```

де в якості `імені_заголовного_файлу` підставляються:

```
limits.h - для даних цілих типів;
float.h- для дійсних даних.
```

У заголовок **`limits.h`** автори компілятора помістили набір

препроцесорних директив, серед яких є такі (значення наведені в шістнадцятковому вигляді):

```
#define        CHAR_BIT       8
#define        SHRT_MAX       0x7FFF
#define        LONG_MAX       0x7FFFFFFF
```

У заголовок **float.h** знаходяться директиви, що визначають константи, які пов'язані з представленням даних дійсних типів. Наприклад:

```
#define        FLT_MIN        1.17549435E-38F
#define        DBL_MIN        2.2250738585072014E-308
#define        DBL_EPSILON    2.220446049250313lE-16
```

Значення цих зумовлених на препроцесорному рівні констант відповідно до стандарту мови в конкретних компіляторах можуть бути дещо іншими.

Зараз досить знати те, що, записавши в тексті своєї програми директиву `#include <limits.h>` можна використовувати в програмі стандартні іменовані константи **CHAR_BIT, SHRT_MIN** та інші, а вже їх значеннями будуть ті числа, які включили до директив `#define` автори конкретного компілятора та конкретної бібліотеки.

Якщо включити в програму директиву `#include <float.h>` то стануть доступними іменовані константи граничних значень числових даних дійсних типів.

Такий підхід до визначення граничних значень за допомогою препроцесорних констант, що зберігаються в бібліотечних файлах, дозволяє писати програми, які не залежать від реалізації, що забезпечує їх достатню мобільність. Програміст використовує в програмі стандартні імена (позначення) констант, а їх значення визначаються версією реалізації, тобто конкретним компілятором і його бібліотеками.

3.3. Операції та вирази мови C

3.3.1. Знаки операцій

Для формування та подальшого обчислення виразів використовуються операції. Для зображення однієї операцій у більшості випадків використовується кілька символів. У табл. 3.6 наведені всі знаки операцій, які задані стандартом мови **C**. Операції в таблиці розбиті на групи відповідно до їх рангів.

За винятком операцій «**[]**», «**()**» і «**?:**», усі знаки операцій розпізнаються компілятором як окремі лексеми. Залежно від контексту одна й та ж сама лексема може позначати різні операції, тобто один і той самий знак операції може вживатися в різних виразах і по-різному інтерпретуватися в залежності від контексту. Наприклад, бінарна операція **&** – це порозрядна кон’юнкція, а унарна операція **&** – це операція отримання адреси.

Таблиця 3.6 – Пріоритети (ранги) операцій

Ранг	Операції	Асоціативність
1.	**() [] ->**	→
2.	**! ~ + - ++ -- & * (тип) sizeof** (унарні)	←
3.	*** / %** (мультиплікативні бінарні)	→
4.	**+ -** (адитивні бінарні)	→
5.	**<< >>** (порозрядного зсуву)	→
6.	**< <= > =>** (відношення)	→
7.	**== !=** (відношення)	→
8.	**&** (порозрядна кон’юнкція «І»)	→
9.	**^** (порозрядна виключна диз’юнкція або додавання за модулем 2)	→
10.	**¦** (порозрядна диз’юнкція «АБО»)	→
11.	**&&** (кон’юнкція «І»)	→
12.	**¦¦** (диз’юнкція «АБО»)	→
13.	**?:** (умовна операція)	←
14.	**= *= /= %= += -= &= ^= ¦= <<= >>=**	←
15.	**,** (операція «кома»)	→

Пояснення до таблиці 3.6:

1) Операції рангу 1 мають найвищий пріоритет.

2) Операції одного рангу мають однаковий пріоритет, і якщо їх у виразі кілька, то вони виконуються відповідно до правил асоціативності або зліва направо (→), або справа наліво (←).

3) Якщо один і той самий знак операції наведено в таблиці двічі (наприклад, знак `*`), то перша поява (з меншим за номером, тобто старшим за пріоритетом, рангом) відповідає унарній операції, а друга – бінарній.

Опишемо коротко можливості окремих операцій.

Для зображення одномісних префіксних і постфіксних операцій використовуються такі символи:

& – *операція отримання адреси* операнда (ранг 2);

***** – *операція звернення за адресою*, тобто розкриття посилання, інакше операція розіменування (доступу за адресою до значення того об'єкта, на який вказує операнд). Операндом повинен бути вказівник (ранг 2);

– – *унарний мінус*, змінює знак арифметичного операнда (ранг 2);

+ – *унарний плюс*, введений для симетрії з унарним мінусом (ранг 2);

~ – *порозрядне інвертування* внутрішнього двійкового коду цілочислового аргументу – *побітове заперечення* (ранг 2);

! – НЕ – *логічне заперечення* значення операнда (ранг 2). Застосовується до скалярних операндів. Цілочисловий результат `0` (якщо операнд ненульовий, тобто істинний) або `1` (якщо операнд нульовий, тобто помилковий). Слід нагадати, що в якості логічних значень у мові **C** використовують цілі числа: `0` – хибність і не нуль, тобто (`!0`) – істина. Запереченням будь-якого ненульового числа буде `0`, а запереченням нуля буде `1`. Таким чином: `!1` дорівнює `0`; `!2` дорівнює `0`; `!(-5)` дорівнює `0`; `!0` дорівнює `1`;

++ – *збільшення на одиницю* (*інкремент* – ранг 2), має дві форми:

1) *префіксна операція* – збільшення значення операнда на `1` до його використання. Асоціативність справа відповідно до стандарту;

2) *постфіксна операція* – збільшення значення операнда на **1** після його використання. Асоціативність зліва відповідно до стандарту.

Операнд для операції **++** (і для операції **--**) не може бути константою або довільним виразом. Записи **++5** або **84++** будуть невірними. **++(j+k)** також невірний запис.

Операндами унарних операцій **++** і **--** повинні бути завжди **l**-виразами. Термін «**l**-вираз» походить від пояснення дії операції присвоювання **E = D**, в якій операнд **E**, який знаходиться зліва від знака операції присвоювання, може бути тільки **l**-виразом, що модифікується. Прикладом **l**-виразу, що модифікується, є ім'я змінної, якій виділена пам'ять.

Таким чином, **l**-вираз – це посилання на область пам'яті, значення якої можна змінювати;

-- – *зменшення на одиницю* (*декремент* – ранг 2) – унарна операція, операндом якої повинен бути **l**-вираз, тобто не константа і не вираз:

1) *префіксна операція* – зменшення на **1** значення операнда до його використання;

2) *постфіксна операція* – зменшення на **1** значення операнда після його використання;

sizeof – *операція* (ранг 2) *обчислення розміру* (в байтах) для об'єкта того типу, який має операнд. Дозволені два формати операції:

```
sizeof вираз
sizeof (тип)
```

sizeof не обчислює значення виразу, а лише визначає його тип, для якого потім обчислюється розмір.

Бінарні операції поділяються на наступні групи:

- адитивні;

- мультиплікативні; - зсувів;

- порозрядні;

- операції відношень;

- логічні;

- присвоєння;

- вибору компонента структурованого об'єкта; - операція «кома»;

- дужки в якості операцій.

Адитивні операції:

+ – *бінарний плюс* – додавання арифметичних операндів або додавання вказівника до цілочислового операнда (ранг 4);

- – *бінарний мінус* – віднімання арифметичних операндів або віднімання вказівників (ранг 4).

Мультиплікативні операції:

***** – *множення* операндів арифметичного типу (ранг 3);

/ – *ділення* операндів арифметичного типу (ранг 3). За умови цілочислових операндів абсолютне значення результату округляється до цілого. **Наприклад, 20/3** дорівнює **6**, **-20/3** дорівнює **-6**, **(-20)/3** дорівнює **-6**, **20/(-3)** дорівнює **-6**;

% – *отримання залишку від ділення* цілочислових операндів (*ділення за модулем* – ранг 3). За умови невід'ємних операндів – залишок позитивний. В іншому випадку залишок визначається реалізацією:

```
13 % 4 дорівнює 1,
13 % (-4) дорівнює 1,
(-13) % 4 дорівнює -1;
(-13) % (-4) дорівнює -1.
```

При ненульовому дільнику для цілочислових операндів завжди виконується співвідношення: `(a / b) * b + a % b` дорівнює `a`.

Операції зсуву визначені тільки для цілочислових операндів. Формат виразу з операцією зсуву:

```
операнд_лівий операція_зсуву операнд_правий
```

<< – *зсув вліво* бітового представлення значення лівого цілочислового операнда на кількість розрядів, що дорівнює значенню правого цілочислового

операнда (ранг 5);

>> – *зсув вправо* бітового представлення значення лівого цілочислового операнда на кількість розрядів, що дорівнює значенню правого цілочислового операнда (ранг 5).

До *порозрядних операцій* належать:

& – *порозрядна кон'юнкція* (І) бітових представлень значень цілочислових операндів (ранг 8);

¦ – *порозрядна диз'юнкція* (АБО) бітових представлень значень цілочислових операндів (ранг 10);

^ – *порозрядна виключна диз'юнкція* або *додавання за модулем 2* бітових представлень значень цілочислових операндів (ранг 9).

Результат виконання операцій зсуву та порозрядних операцій:

```
4 << 2   дорівнює 16;
5 >> 1   дорівнює 2;
6 & 5    дорівнює 4;
6 ¦ 5    дорівнює 7;
6 ^ 5    дорівнює 3.
```

Слід нагадати, що двійковий код для **4** дорівнює **100**, для **5** – це **101**, для **6** – це **110** і так далі. При зсуві вліво на дві позиції код **100** стає рівним **10000** (десяткове значення дорівнює **16**). Решту результатів операцій зсуву та порозрядних операцій можна простежити аналогічним чином.

Зверніть увагу на те, що зсув вліво на **n** позицій еквівалентний множенню значення на 2^n, а зсув коду вправо зменшує відповідне значення в 2^n разів з відкиданням дробової частини результату (тому **5 >> 1** дорівнює **2**).

До *операцій відношення* належать:

< – менше, ніж (ранг 6);
> – більше, ніж (ранг 6);
<= – менше або дорівнює (ранг 6);
>= – більше або дорівнює (ранг 6);

== – дорівнює (ранг 7);
!= – не дорівнює (ранг 7).

Операнди операцій відношення повинні бути арифметичного типу або можуть бути вказівниками. Результат цілочисловий: **0** (хибність) або **1** (істина). Останні дві операції (операції порівняння на рівність) мають більш низький пріоритет у порівнянні з іншими операціями відношення. Таким чином, вираз (**x < B == A < x**) є **1**, коли значення **x** знаходиться в інтервалі від **A** до **B** і **A < B** або **x**, **A**, **B** є рівними (спочатку обчислюються **x < B** і **A < x**, а до результатів застосовується операція порівняння на рівність **==**).

До *логічних бінарних операцій* належать:

&& – кон'юнкція (І) арифметичних операндів або відношень (ранг 11). Цілочисловий результат **0** (хибність) або **1** (істина);

|| – диз'юнкція (АБО) арифметичних операндів або відношень (ранг 12). Цілочисловий результат **0** (хибність) або **1** (істина).

Результати операцій відношення та логічних операцій:

3 < 5 дорівнює **1**;

3 > 5 дорівнює **0**;

3 == 5 дорівнює **0**;

3 != 5 дорівнює **1**;

3 != 5 || 3 == 5 дорівнює **1**;

3 + 4 > 5 && 3 + 5 > 4 && 4 + 5 > 3 дорівнює **1**.

Операції присвоювання мають ранг 14. В якості лівого операнда в операціях присвоювання може використовуватися тільки **l**-вираз, тобто посилання на деяку іменовану область пам'яті, значення якої може змінюватися.

Перерахуємо операції присвоювання, зазначивши, що існують одна проста операція присвоювання і декілька складених операцій:

= – *просте присвоювання*: присвоїти значення виразу-операнда з правої частини операнду лівої частини. Наприклад: **P = 10.3 - 2 * x**;

***=** – *присвоювання після множення*: присвоїти операнду лівої частини добуток значень обох операндів. Наприклад: **P *= 2** еквівалентно **P = P * 2**;

/= – *присвоювання після ділення*: присвоїти операнду лівої частини частку від ділення значення лівого операнда на значення правого. Наприклад: **P /= 2.2 - d** еквівалентно **P = P / (2.2 - d)**;

%= – *присвоювання після ділення по модулю*: присвоїти операнду лівої частини остачу від цілочислового ділення значення лівого операнда на значення правого операнда. Наприклад: **N %= 3** еквівалентно **N = N % 3**;

+= – *присвоювання після підсумовування*: присвоїти операнду лівої частини суму значень обох операндів. Наприклад: **A += B** еквівалентно **A = A + B**;

-= – *присвоювання після віднімання*: присвоїти операнду лівої частини різницю значень лівого і правого операндів. Наприклад: **X -= 4.3 - Z** еквівалентно **X = X - (4.3 - Z)**;

<<= – *присвоювання після зсуву розрядів вліво*: присвоїти цілочисловому операнду лівої частини значення, отримане зсувом вліво його бітового представлення на кількість розрядів, що дорівнює значенню правого цілочислового операнда. Наприклад: **a <<= 4** еквівалентно **a = a << 4**;

>>= – *присвоювання після зсуву розрядів вправо*: присвоїти цілочисловому операнду лівої частини значення, отримане зсувом вправо його бітового представлення на кількість розрядів, що дорівнює значенню правого цілочислового операнда. Наприклад: **a >>= 4** еквівалентно **a = a >> 4**;

&= – *присвоювання після порозрядної кон'юнкції*: присвоїти цілочисловому операнду лівої частини значення, отримане порозрядною кон'юнкцією (І) його бітового представлення з бітовим представленням цілочислового операнда правої частини. Наприклад: **e &= 44** еквівалентно **e = e & 44**;

¦= – *присвоювання після порозрядної диз'юнкції*: присвоїти

цілочисловому операнду лівої частини значення, отримане порозрядною диз'юнкцією (АБО) його бітового представлення з бітовим представленням цілочислового операнда правої частини. Наприклад: **a |= b** еквівалентно **a = a | b**;

^= – *присвоювання після порозрядної виключної диз'юнкції*: присвоїти цілочисловому операнду лівої частини значення, отримане застосуваннямпорозрядної виключної диз'юнкції до бітових представлень значень обох операндів. Наприклад: **z ^= x + y** еквівалентно **z = z ^ (x + y)**.

Зверніть увагу, що для всіх складених операцій присвоювання форма присвоювання **E1 op= E2** еквівалентна **E1 = E1 op (E2)**, де **op** – позначення операції.

До *операцій вибору компонентів структурованого об'єкта* належать:

. (*крапка*) – *прямий вибір (виділення) компонента структурованого об'єкта*, наприклад, об'єднання або структури (ранг 1).

Формат застосування операції:

```
ім'я_структурованого_об'єкта.ім'я_компонента
```

-> – *непрямий вибір (виділення) компонента структурованого об'єкта*, що адресується вказівником (ранг 1). При використанні операції потрібно, щоб з об'єктом був пов'язаний вказівник. У цьому випадку формат застосування операції має вигляд:

```
вказівник_на_структурований_об'єкт->ім'я_компонента
```

Оскільки операції вибору компонентів структурованих об'єктів використовуються зі структурами та об'єднаннями, то необхідні пояснення та приклади будуть наведені пізніше, коли дані поняття будуть розглянуті.

Кома в якості операції має ранг 15. Кілька виразів, розділених комами «**,**», обчислюються послідовно зліва направо. В якості результату зберігаються тип і значення самого правого виразу. Таким чином, операція «кома» групує обчислення зліва направо. Тип і значення результату визначаються самим правим з розділених комами операндів (виразів). Значення всіх лівих

операндів ігноруються. Наприклад, якщо змінна **x** має тип **int**, то значенням виразу (**x = 3, 3 * x**) буде **9**, а змінна **x** прийме значення **3**.

Дужки в якості операцій. Круглі **()** і квадратні **[]** дужки відіграють роль бінарних операцій (ранг 1) під час виклику функцій та індексування елементів масивів. Для програміста, який тільки розпочинає використовувати мову **C**, думка про те, що дужки в ряді випадків є бінарними операціями, часто навіть не приходить в голову. І це навіть тоді, коли він практично в кожній програмі звертається до функцій або застосовує індексовані змінні. Слід зазначити, що дужки можуть служити бінарними операціями, особливості та можливості яких варті уваги.

Круглі дужки обов'язкові у зверненні до функції:

```
ім'я_функції(список_аргументів)
```

де операндами служать `ім'я_функції` і `список_аргументів`. Результат виклику визначається (обчислюється) в тілі функції, структуру якого задає її визначення.

У виразі

```
ім'я_масиву[індекс]
```

операндами для операції **[]** служать `ім'я_масиву` та `індекс`.

Умовна тримісна операція має ранг 13. На відміну від унарних і бінарних операцій умовна тернарна операція використовується з трьома операндами. У зображенні умовної операції застосовуються два символи **'?'** та **':'** і три вирази-операнди:

```
вираз_1 ? вираз_2 : вираз_3
```

Першим обчислюється значення **виразу_1**. Якщо він істинний, тобто

не дорівнює нулю, то обчислюється значення **виразу_2**, який стає результатом. Якщо при обчисленні **виразу_1** отримаємо **0**, то в якості результату береться значення **виразу_3**. Класичний приклад:

```
x < 0 ? -x : x;
```

Вираз повертає абсолютну величину змінної x.

Операція явного перетворення (приведення) типу (ранг 2) має такий вигляд:

```
(ім'я_типу) операнд
```

Такий вираз дозволяє перетворювати значення операнда до заданого типу. В якості операнда використовується унарний вираз, який в найпростішому випадку може бути змінною, константою або будь-яким виразом в круглих дужках. Наприклад, перетворення **(long)8** (внутрішнє представлення результату має довжину **4** байта) і **(char)8** (внутрішнє представлення результату має довжину **1** байт) змінюють довжину внутрішнього представлення цілих констант, не змінюючи їх значень.

У цих перетвореннях константа не міняла значення і залишалася цілочисловою. Однак можливі більш глибокі перетворення, наприклад, **(long double)6** або **(float)4** не тільки змінюють довжину константи, але й структуру її внутрішнього представлення. У результатах будуть виділені порядок і мантиса, значення будуть дійсними.

Перетворення типів арифметичних даних потрібно застосовувати акуратно, оскільки можлива зміна числових значень. При перетворенні великих цілочислових констант до дійсного типу (наприклад, до типу **float**) можлива втрата значущих цифр (втрата точності). Якщо дійсне значення перетвориться до цілого, то можлива помилка при виході отриманого значення за діапазон допустимих значень для цілих. В цьому випадку

результат перетворення не завжди передбачуваний і цілком залежить від реалізації.

Роздільники. Повнота викладу відомостей про лексеми та їх призначення вимагає систематичного розгляду роздільників.

Роздільники, або знаки пунктуації, входять до числа лексем мови:

[]   ()   {}   ,   ;   :   ...   *   =   #

Квадратні дужки **[]** використовуються для обмеження індексів одно- і багатовимірних масивів, а також під час запису індексованих елементів.

Приклади:

```
int A[5];        // A - одновимірний масив з п'яти елементів
int x, e[3][2]; // e - двовимірний масив (матриця) розміром 3x2
```

Вираз з індексованими елементами:

```
e[0][0] = x = A[2] = 4;
```

означає, що початковому елементу масиву **e**, змінної **x** і третього елементу масиву **A** присвоюється значення **4**. Оскільки індекси в масивах завжди починаються з **0**, то елемент **A[2]** відповідає третьому елементу масиву.

Призначення *круглих дужок* **()**:

1) виділяють вирази-умови (в операторі **if**):

```
// абсолютна величина арифметичної змінної
if(x < 0) x = -x;
```

2) входять як обов'язкові елементи у визначення та опис (в прототип) будь-якої функції, де виділяють відповідно список формальних параметрів і список специфікацій параметрів:

```
float F(float x, int k)  // Визначення функції
{
```

```
      тіло_функції
}
float F(float, int);     // Опис функції - її прототип
```

3) круглі дужки є обов'язковими при визначенні вказівника на функцію:

```
int (*pfunc)();     // Визначення вказівника pfunc на функцію
```

4) групують вирази, змінюючи природну послідовність виконання операцій:

```
y = (a + b) / c;    // Зміна пріоритету операції
```

5) входять як обов'язкові елементи в оператори циклів:

```
for(i = 0, j = 1; i < j; i += 2, j++)тіло_цикла;
while (i < j) тіло_цикла;
do тіло циклу while (k > 0);
```

6) необхідні при явному перетворенні типу. Приклади:

```
long i = 12L;     // Визначення змінної i
float brig;       // Визначення змінної brig
brig = (float)i; // Явне приведення типу
```

`brig` отримує значення `12L`, перетворене до типу **`float`**.

7) застосування круглих дужок настійно рекомендується в макровизначеннях, які обробляються препроцесором:

```
#define  R(x, y)  sqrt((x) * (x) + (y) * (y))
```

Це дозволяє використовувати в якості параметрів макровикликів арифметичні вирази будь-якої складності та не стикатися з порушеннями

пріоритетів операцій.

Фігурні дужки. Для позначення відповідно початку і кінця складеного оператора або блоку використовують фігурні дужки **{}**. Приклад використання складеного оператора в умовному операторі:

```
if(d >x)
{
      d--;
      x++;
}
```

Приклад блоку – тіло будь-якої функції:

```
float absx(float x)
{
      return x > 0.0 ? x : x;
}
```

Зверніть увагу на відсутність крапки з комою після закривальної дужки **'}'**, що позначає кінець складеного оператора або блоку.

Фігурні дужки використовуються для виділення списку компонентів в визначеннях типів структур і об’єднань:

```
// Визначення типу структура cell
struct cell
{
      char        *b;
      int         ee;
      double      U[6]
};

// Визначення типу об'єднання mix
union mix
{
      unsigned int    ii;
      char            cc[2];
};
```

Зверніть увагу на необхідність крапки з комою після визначення кожного типу.

Фігурні дужки використовуються при ініціалізації масивів і структур при їх визначенні:

```
// Ініціалізація масиву
int month[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };

// Ініціалізація структури stock типу mixture
struct mixture
{
int             ii;
double          dd;
char            cc;
} Stock = { 666, 3.67, '\t' };
```

У прикладі **mixture** – ім'я структурного типу з трьома компонентами різних типів, **stock** – ім'я конкретної структури типу **mixture**. Компоненти **ii**, **dd**, **cc** структури **stock** отримують значення при ініціалізації зі списку в фігурних дужках.

Кома може бути використана в якості операції, а може застосовуватися як роздільник. В останньому випадку вона розділяє елементи списків початкових значень елементів масивів і компонентів структур при їх ініціалізації.

Інший приклад списків – списки формальних і фактичних параметрів і їх специфікацій у функціях.

Третє використання коми як роздільник – в заголовку оператора циклу:

```
for(x = p1, y * p2, i = 2; i < n; z = x+y, x = y, y = z, i++);
```

В даному прикладі після виконання циклу значенням змінної **z** буде величина, що дорівнює **n**-му члену послідовності чисел Фібоначчі, яка визначається за значенням перших двох **p1** і **p2**.

Кома як роздільник використовується також в описах і визначеннях об'єктів (наприклад, змінних) одного типу:

```
int       i, n;
float     x, y, z, p1, p2;
```

Кома в якості операції вже розглядалася. Слід звернути увагу на необхідність за допомогою круглих дужок відокремлювати кому-операцію

від коми-роздільника. Наприклад, для елементів наступного масиву **m** використовується список з трьома початковими значеннями:

```
int i = 1, m[] = { i, (i = 2, i * i), i};
```

В даному прикладі кома в круглих дужках виступає в ролі знака операції. Операція присвоювання **«=»** має більш високий пріоритет, ніж операція «кома». Тому спочатку **i** отримує значення **2**, потім обчислюється добуток **i * i**, і цей результат є значенням виразу в дужках. Однак значенням змінної **i** залишається **2**. Значеннями **m[0]**, **m[1]**, **m[2]** будуть відповідно **1**, **4**, **2**.

Крапка з комою. Кожен оператор, кожне визначення і кожен опис у програмі на мові **C** завершує крапка з комою **';'**. Будь-який допустимий вираз, за яким іде **';'**, сприймається як оператор. Це справедливо і для порожнього виразу, тобто окремий символ «крапка з комою» вважається порожнім оператором. Порожній оператор іноді використовується як тіло циклу. Прикладом може служити цикл **for**, наведений вище для ілюстрації особливостей використання коми в якості роздільника (обчислюється **n**-й член послідовності чисел Фібоначчі).

Приклади операторів-виразів:

```
i++;        // Результат- лише зміна значення змінної i
F(z, 4);    // Результат визначається тілом функції з ім'ям F
```

Двокрапка. Для відділення мітки від оператора, що позначається нею, використовуєтьсядвокрапка **':'**:

```
мітка: оператор;
```

Три крапки **'...'**без пробілів між ними використовується для позначення змінного числа параметрів у функції при її визначенні та описі (при задаванні її прототипу). При роботі на мові **C**, програміст постійно використовує бібліотечні функції зі списком параметрів змінної довжини для форматних введення і виведення. Їх прототипи виглядають наступним чином:

```
int printf(const char * format,...);
int scanf(const char * format, ...);
```

Тут за допомогою трьох крапок вказана можливість при зверненні до функцій використовувати різну кількість параметрів (не менш одного, оскільки параметр `format` повинен бути вказаний завжди і не може опускатися).

Підготовка своїх функцій зі змінною кількістю параметрів на мові **C** вимагає застосування засобів адресної арифметики, наприклад макроси, що надаються заголовним файлом `stdarg.h`.

Зірочка. Як вже згадувалося, зірочка `'*'` використовується в якості знака операції множення і знака операції розіменування (отримання доступу через вказівник). В описах і визначеннях зірочка означає, що описується (визначається) вказівник на значення типу, який був використаний в оголошенні:

```
// вказівник на величину типу int
int       *point;

// вказівник на вказівник на об'єкт типу char
char      **refer;
```

Позначення присвоювання. Як вже згадувалося, для позначення операції присвоювання використовується символ `'='`. Крім того, у визначенні об’єкта він використовується при його ініціалізації:

```
// ініціалізація структури
struct
{
     char x, int y
} A = { 'z', 1918 };

// ініціалізація змінної
int F = 66;
```

Ознака директиви препроцесора. Символ `'#'` (знак номера або дієза в музиці) використовується для позначення директив (команд) препроцесора. Якщо цей символ є першим відмінним від пробілу символом у рядку програми, то рядок сприймається як директива препроцесора. Цей же символ використовується в якості однієї з препроцесорних операцій.

Без однієї з препроцесорних директив обійтися абсолютно неможливо – препроцесорна команда `#include <stdio.h>` включає в текст програми засоби зв'язку з бібліотечними функціями вводу-виводу.

Директива `#define` вже була введена при розгляді іменованих препроцесорних констант. Решта директив і операцій препроцесора будуть розглядатися пізніше.

3.3.3. Вирази та приведення арифметичних типів

Ввівши константи, змінні, роздільники та знаки операцій, охарактеризувавши основні типи даних і розглянувши змінні, можна конструювати вирази. Кожен вираз складається з одного або декількох операндів, символів операцій і обмежувачів, в якості яких найчастіше виступають круглі дужки `()`. Призначення будь-якого виразу – формування деякого значення. Залежно від типу значень, що формуються, визначаються типи виразів. Якщо значеннями виразу є цілі та дійсні числа, то говорять про арифметичні вирази. В арифметичних виразах припустимі наступні операції:

+ – додавання (або унарна операція **+**);

- – віднімання (або унарна операція зміни знака); ***** – множення;

/ – ділення;

% – ділення по модулю (тобто отримання залишку від цілочислового ділення першого операнда на другий).

Операндами для перерахованих операцій служать константи та змінні арифметичних типів, а також вирази в круглих дужках.

Приклади виразів з двома операндами:

```
a+b       12.3-x    3.14159*Z     k/3       16%i
```

Потрібно бути акуратним, застосовуючи операцію ділення «**/**» до цілочислових операндів. За рахунок округлення результату значенням виразу **5/3** буде **1**, а чи відповідає це задумам програміста, залежить від

змісту тієї конкретної конструкції, в якій цей вираз використовується.

Щоб результат виконання арифметичної операції був дійсним, необхідно, щоб дійсними був хоча б один з операндів. Наприклад, значенням виразу **5.0 / 2** буде **2.5**, що відповідає змісту звичайного ділення.

Операції *, /, % мають один ранг (3), операції +, – також один ранг (4), але більш низький. Арифметичні операції одного рангу виконуються зліва направо. Для зміни порядку виконання операцій зазвичай використовуються дужки. Наприклад, вираз **(d + b) / 2.0** дозволяє отримати середнє арифметичне операндів **d** і **b**.

Як вже говорилося, в **C** введені специфічні унарні операції **++** (інкремент) та **--** (декремент) для зміни на **1** операнда, який в найпростішому випадку повинен бути змінною (l-виразом). Кожна з цих операцій може бути префіксною і постфіксною:

- вираз **++m** збільшує на **1** значення **m**, і це отримане значення використовується як значення виразу **++m** (префіксна форма);

- вираз **--k** зменшує на **1** значення **k**, і це нове значення використовується як значення виразу **--k** (префіксна форма);

- вираз **i++** (постфіксна форма) збільшує на **1** значення **i**, проте значенням вирази **i++** є попереднє значення **i** (до його збільшення);

- вираз **j--** (постфіксна форма) зменшує на **1** значення **j**, однак значенням виразу **j--** є попереднє значення **j** (до його зменшення).

Наприклад, якщо **n** дорівнює **4**, то при обчисленні виразу **n++*2** результат дорівнює **8**, а **n** прийме значення **5**. При обчисленні виразу **++n*2** (якщо **n** дорівнює **4**) результат буде дорівнювати **10**, а **n** стане дорівнювати **5**.

Зовнішню неоднозначність мають вирази, в яких знак унарної операції **++** (або **--**) записаний безпосередньо поруч зі знаком бінарної операції **+**:

```
x+++b або z---d
```

У цих випадках трактування виразів є однозначним і повністю визначається рангами операцій (бінарні адитивні (**+** і **-**) мають ранг 4; унарні (**++** і **--**) мають ранг 2). Таким чином:

`x +++ b` еквівалентно `(x++) + b`
`z --- d` еквівалентно `(z--) - d`

3.3.4. Відношення та логічні вирази

Відношення визначається як пара арифметичних виразів, з'єднаних (розділених) знаком операції відношення. Знаки операцій відношення (вже були введені вище):

== – дорівнює;

!= – не дорівнює;

< – менше, ніж;

<= – менше або дорівнює;

> – більше, ніж;

>= – більше або дорівнює.

Приклади відношень:

```
a - b > 6.3
(X - 4) * 3 == 12
6 <= 44
```

Відношення має не нульове значення (зазвичай **1**), якщо воно істинне, і дорівнює **0**, якщо воно помилкове. Таким чином, значенням відношення **6 <= 44** буде **1**.

Операції **>**, **>=**, **<**, **<=** мають один ранг 6. Операції порівняння на рівність (**==** і **!=**) також мають однаковий, але нижчий ранг 7, ніж інші операції відношення. Арифметичні операції мають більш високий ранг, ніж операції відношення, тому в першому прикладі для виразу **a-b** не потрібні дужки.

Логічних операцій у мові **C** три:

! – заперечення, тобто логічне НЕ (ранг 2);

&& – кон'юнкція, тобто логічне І (ранг 11);

`||` – диз’юнкція, тобто логічне АБО (ранг 12).

Вони перераховані за спадним старшинством (рангом). Як правило, логічні операції застосовуються до відношень. До виконання логічних операцій обчислюються значення відношень, що входять до логічного виразу. Наприклад, якщо **`a`**, **`b`**, **`c`** – змінні, відповідні довжини сторін трикутника, то для них має бути істинним, тобто таким, що не дорівнює **`0`**, наступний логічний вираз:

```
a + b > c && a + c > b && b + c > a
```

Кілька операцій одного рангу виконуються зліва направо, причому обчислення перериваються, як тільки буде визначена істинність (або хибність) результату, тобто якщо в розглянутому прикладі **`a + b`** виявиться не більше за **`c`**, то інші відношення не розглядаються – результат хибність.

Оскільки значенням відношення є ціле (**`0`** або **`1`**), то ніщо не суперечить застосуванню логічних операцій до цілочислових значень. При цьому прийнято, що будь-яке ненульове позитивне значення сприймається як істинне, а помилковою вважається тільки величина, що дорівнює нулю. Значенням **`!5`** буде **`0`**, значенням **`4 && 2`** буде **`1`** і так далі.

3.3.5. Присвоювання (вираз і оператор)

Як вже говорилося, символ «**`=`**» в мові **`C`** позначає бінарну операцію, в якій у виразі повинно бути два операнда – лівий (**`l`**-вираз – зазвичай змінна) і правий (зазвичай вираз). Якщо **`z`** – ім’я змінної, то

```
z = 2.3 + 5.1
```

є вираз із значенням **`7.4`**. Одночасно це значення присвоюється і змінній **`z`**. Тільки в тому випадку, коли в кінці виразу з операцією присвоювання поміщений символ «**`;`**» цей вислів стає оператором присвоювання. Таким чином,

```
z = 2.3 + 5.1;
```

є оператором простого присвоювання змінній **`z`** значення, що дорівнює **`7.4`**.

Тип і значення виразу з операцією присвоювання визначаються значенням виразу, який розміщується праворуч від знака **«=»**. Однак цей тип може не збігатися з типом змінної з лівої частини виразу. В цьому випадку при визначенні значення змінної виконується перетворення (приведення) типів.

Оскільки вираз праворуч від знака `'-'` може містити, у свою чергу, операцію присвоювання, то в одному операторі присвоювання можна присвоювати значення декільком змінним, тобто організувати «множинне» присвоювання, наприклад:

```
c = x = d = 4.0 + 2.4;
```

В даному операторі значення **6.4** присвоюється змінній **d**, потім **6.4** як значення виразу з операцією присвоювання «**d = 4.0 + 2.4**» присвоюється **x** і, нарешті, **6.4** як значення виразу «**x = d**» присвоюється змінній **c**. Природне обмеження – зліва від знака **«=»** в кожній з операцій присвоювання може бути тільки **l**-вираз.

У мові **C** існує цілий набір «складених операцій присвоювання». Кожна із складених операцій присвоювання об'єднує деяку бінарну логічну або арифметичну операцію і власне саме присвоювання. Операція складеного присвоювання є основою оператора складеного присвоювання:

```
ім'я_змінної op= вираз;
```

де **op** – одна з операцій ***, /, %, +, -, &, ^, |, <<, >>**. Якщо розглядати конструкцію **«op=»** як дві операції, то спочатку виконується **op**, а потім **'='**. наприклад,

```
x *= 2;          z += 4;          i /= x + 4 * z;
```

При виконанні кожного з цих операторів операндами для операції **op** служать змінна з лівої частини і вираз з правої. Результат присвоюється змінній з лівої частини.

Таким чином, перший приклад можна розглядати як позначення вимоги «подвоїти значення змінної **x**»; другий приклад – «збільшити на **4** значення змінної **z**»; третій приклад – «зменшити значення змінної **i** в **(x + 4 ***

z) разів». Цим операторам еквівалентні такі оператори простого присвоювання:

```
x = x * 2;          z = z + 4;          i = i / (x + 4 * z);
```

В останньому з них довелося ввести дужки для отримання правильного результату. Зверніть увагу на те, що перейти від простого оператора присвоювання до складеного можна тільки в тих випадках, коли одна змінна використовується в обох частинах. Більш того, для деяких операцій ця змінна повинна бути обов'язково першим (лівим) операндом. Наприклад, на вдасться замінити складовими наступні прості оператори присвоювання:

```
a = b / a;          x = z % x;
```

3.3.6. Приведення типів

Розглядаючи операцію ділення, було відзначено, що при діленні двох цілих операндів результат виходить цілим. Наприклад, значенням виразу **5/2** буде **2**, а не **2.5**. Для отримання дійсного результату потрібно виконувати ділення не цілих, а дійсних операндів, наприклад, записавши **5.0/2.0**, отримаємо значення **2.5**.

Якщо операндами є безіменні константи, то замінити цілу константу на дійсну зовсім не важко. У тому випадку, коли операндом є іменована константа, змінна або вираз в дужках, необхідно для вирішення тієї ж задачі використовувати операцію явного приведення (перетворення) типу. Наприклад, розглянемо такий набір визначень і операторів присвоювання:

```
int       n = 5, k = 2;
double    d;
int       m;

d = (double)n / (double)k;
m = n / k;
```

У цьому фрагменті значенням **d** стане величина **2.5** типу **double**, а значенням змінної **m** стане ціле значення **2**.

Операція ділення є тільки однією з бінарних операцій. Майже для кожної

з них операнди можуть мати різні типи. Однак не завжди програміст повинен в явному вигляді вказувати перетворення типів. Якщо у бінарної операції операнди мають різні типи (а повинні відповідно до синтаксису виразу мати один тип), то компілятор виконує перетворення типів автоматично, тобто приводить обидва операнда до одного типу. Наприклад, для тих же змінних значення виразу `d + k` матиме тип `double` за рахунок неявного перетворення, яке виконується автоматично без вказівки програміста. Розглянемо правила, за якими такі приведення виконуються.

Правила перетворення типів

При обчисленні виразів деякі операції вимагають, щоб операнди мали відповідний тип, а якщо вимоги до типу не виконані, примусово викликають виконання потрібних перетворень. Така ж ситуація виникає при ініціалізації, коли тип виразу, який ініціалізується, приводиться до типу об'єкта, що визначається. Згадаємо, що в мові `C` присвоювання є бінарної операцією, тому сказане щодо перетворення типів відноситься і до всіх форм присвоювання, однак при присвоюванні значення виразу з правої частини завжди приводиться до типу змінної з лівої частини, незалежно від співвідношення цих типів.

Правила перетворення в мові Сдля основних типів визначені стандартом. Ці стандартні перетворення включають переведення «нижчих» типів до «вищих».

Серед перетворень типів виділяють:

- перетворення в арифметичних виразах; - перетворення при присвоюванні;

- перетворення вказівників.

Перетворення типів вказівників буде розглянуто наступному розділі.

Зараз розглянемо перетворення типів при арифметичних операціях і особливості перетворень типів при присвоюванні.

При перетворенні типів потрібно розрізняти перетворення, які змінюють внутрішнє представлення даних, і перетворення, які змінюють тільки інтерпретацію внутрішнього представлення. Наприклад, коли дані типу

unsigned int переводяться в тип int, міняти їх внутрішнє представлення не потрібно – змінюється тільки інтерпретація. При перетворенні значень типу float в значення типу int недостатньо змінити тільки інтерпретацію, необхідно змінити довжину комірки пам'яті для внутрішнього представлення та кодування. При такому перетворенні з float в int можливий вихід за діапазон допустимих значень типу int, і реакція на цю ситуацію істотно залежить від конкретної реалізації. Саме тому для збереження мобільності програм в них рекомендується з обережністю застосовувати перетворення типів.

Розглянемо послідовність виконання перетворення операндів в арифметичних виразах.

1) Усі короткі цілі типи перетворюються в типи не меншої довжини відповідно до табл. 3.7. Потім обидва значення, які беруть участь в операції, приймають однаковий тип відповідно до наступних простих вказівок.

2) Якщо один з операндів має тип long double, то другий теж буде перетворений в long double.

3) Якщо п. 2 не виконується і один з операндів є double, інший приводиться до типу double.

Таблиця 3.7 – Правила стандартних арифметичних перетворень

Попередній тип	Перетворений тип	Правила перетворення
char	int	Розширення нулем або знаком в залежності від замовчування для char
unsigned char	int	Старший байт заповнюється нулем
signed char	int	Розширення знаком
short	int	Зберігається те ж саме значення
unsigned	unsigned int	Зберігається те ж саме значення
enum	int	Зберігається те ж саме значення
битовое поле	int	Зберігається те ж саме значення

4) Якщо п. 2 – 3 не виконуються і один з операндів має тип float, то другий приводиться до типу float.

5) Якщо п. 2 – 4 не виконуються (обидва операнда цілі) і один операнд

unsigned long int, то обидва операнда перетворюються до типу **unsigned long int**.

6) Якщо п. 2 – 5 не виконуються і один операнд є **long**, інший перетвориться до типу **long**.

7) Якщо п. 2 – 6 не виконуються і один операнд **unsigned**, то інший перетвориться до типу **unsigned**.

8) Якщо п. 2 – 7 не виконані, то обидва операнда належать типу **int**. Використовуючи арифметичні вирази, слід враховувати наведені правила і не потрапляти в «пастки» перетворення типів, оскільки деякі з них призводять до втрат інформації, а інші змінюють інтерпретацію бітового (внутрішнього) представлення даних.

На рис. 3.1 стрілками позначені «безпечні» арифметичні перетворення, що гарантують збереження точності та незмінність числового значення. При перетвореннях, які не віднесені схемою (рис. 6.1) до безпечних, можливі суттєві інформаційні втрати. Для оцінки значущості таких втрат рекомендується перевірити оборотність перетворення типів. Перетворення цілочислових значень в дійсні здійснюється настільки точно, наскільки це передбачено апаратурою. Якщо конкретне цілочислове значення не може бути точно представлено як дійсне, то молодші значущі цифри втрачаються і оборотність неможлива.

Приведення дійсного значення до цілого типу виконується за рахунок відкидання дробової частини. Перетворення цілої величини в дійсну також може призвести до втрати точності.

3.3.7. Вирази з порозрядними операціями

Порозрядні операції дозволяють конструювати вирази, в яких обробка операндів виконується на бітовому рівні (порозрядно). Розглянемо можливості операцій над бітами:

~ – порозрядне заперечення (доповнення або інвертування бітів) (ранг 2);

>> – зсув вправо послідовності бітів (ранг 5);

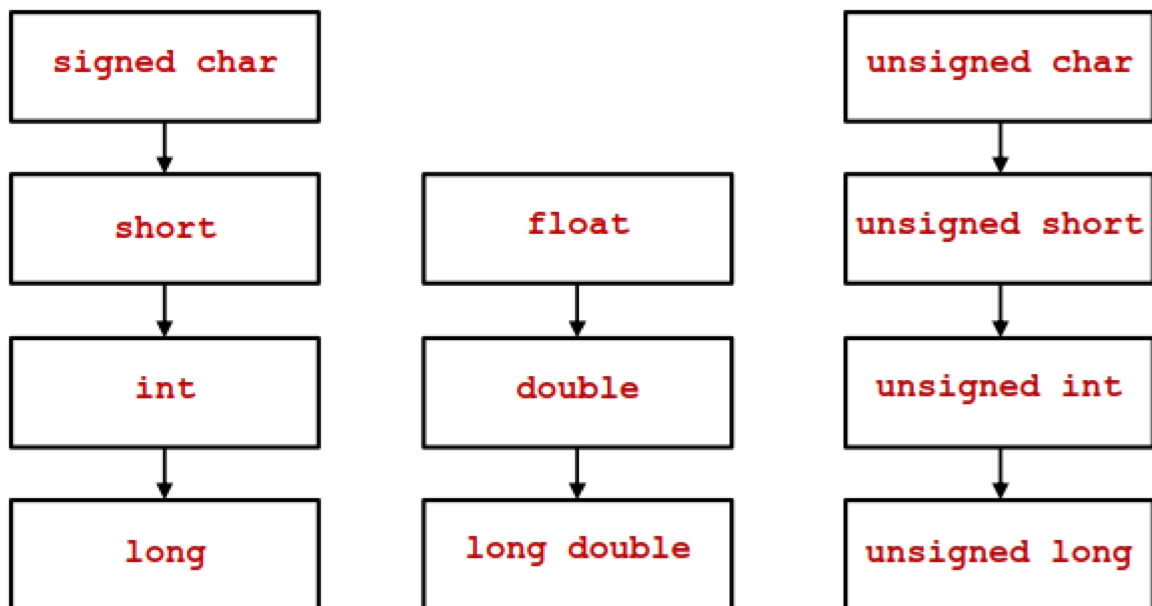


Рисунок 3.1 – Арифметичні перетворення типів, які гарантують збереженість значимості

<< – зсув вліво послідовності бітів (ранг 5);

^ – порозрядна виключна диз’юнкція (ранг 9);

| – порозрядне АБО (порозрядна диз’юнкція) (ранг 10);

& – порозрядне І (порозрядна кон’юнкція) (ранг 8).

Операція порозрядного заперечення (доповнення або інвертування бітів) позначається символом «~» і є унарною (одномісною), тобто діє на один операнд, який повинен бути цілого типу. Значення операнда у вигляді внутрішнього бітового представлення обробляється таким чином, що формується значення тієї ж довжини (того ж типу), що й операнд. У бітовому представленні результату містяться **1** у всіх розрядах, де у операнда **0**, і **0** в тих розрядах, де у операнда **1**.

Приклад:

```
unsigned char E = '\0301', F; F = ~ E;
```

Значенням **F** буде вісімковий код **'\076'** символу **'>'**. Дійсно, бітові представлення значень **E** і **F** можна зобразити так:

11000001 – для значення змінної **E**, тобто для **'\0301'**;

00111110 – для значення змінної **F**, тобто для **'\076'**.

За винятком доповнення, всі інші порозрядні операції бінарні (двомісні).

Операції зсувів **>>** (вправо) і **>>** (вліво) повинні мати цілочислові операнди. Над двійковим представленням значення лівого операнда виконується дія – зсув. Правий операнд визначає величину порозрядного зсуву. Наприклад:

```
5 << 2 дорівнює 20
5 >> 2 дорівнює 1
```

Бітові представлення цих операцій зсуву можна зобразити так:

```
101 << 2 дорівнює 10100, тобто 20;
101 >> 2 дорівнює 001, тобто 1.
```

При зсуві вліво на **N** позицій двійкове представлення лівого операнда зсувається, а розряди, які звільняються справа, заповнюються нулями. Такий зсув еквівалентний множенню значення операнда на 2^N.

Зсув вправо на **N** позицій відбувається дещо складніше. Тут слід зазначити дві особливості. Перша – це зникнення молодших розрядів, що виходять за розрядну сітку. Друга особливість – відсутність стандарту на правило заповнення лівих розрядів, які звільняються. У стандарті мови сказано, що коли лівий операнд є цілим значенням з негативним знаком, то при зсуві вправо заповнення лівих розрядів, які звільняються, визначається реалізацією. Тут можливі два варіанти: звільнені розряди заповнюються значеннями знакового розряду (арифметичний зсув вправо) або звільнені зліва розряди заповнюються нулями (логічний зсув вправо).

При позитивному лівому операнді зсув вправо на **N** позицій еквівалентний зменшенню значення лівого операнда в 2^N разів з відкиданням дробової частини результату (тому **5 >> 2** дорівнює **1**).

Операція «порозрядна виключна диз'юнкція» може бути застосована до цілих операндів. Результат формується при порозрядній обробці бітових

кодів операндів. У тих розрядах, де обидва операнди мають однакові двійкові значення (`1` і `1` або `0` і `0`), результат приймає значення `0`. У тих розрядах, де біти операндів не збігаються, результат дорівнює `1`. Приклад використання:

```
char a = 'A';   // внутрішній код    01000001
char z = 'Z';   // внутрішній код    01011010
a = a ^ z;      // результат:        00011011
z = a ^ z;      // результат:        01000001
a = a ^ z;      // результат:        01011010
```

Змінні `a` і `z` «обмінялися» значеннями без використання допоміжної змінної!

Порозрядна диз'юнкція (порозрядне АБО) може бути застосована до цілочислових операндів. Відповідно до назви вона дозволяє отримати `1` в тих розрядах результату, де не одночасно дорівнюють `0` біти обох операндів.

Наприклад:

```
5 | 6 дорівнює 7 (для 5 - код 101, для 6 - код 110);
10 | 8 дорівнює 10 (для 10- код 1010, для 8 – код 1000).
```

Порозрядна кон'юнкція (порозрядне І) може бути застосована до цілочислових операндів. У бітовому представленні результату тільки ті біти дорівнюють `1`, яким відповідають поодинокі біти обох операндів. Приклади:

```
5 & 6 дорівнює 4 (для 5 – код 101, для 6 – код 110);
10 & 8 дорівнює 8 (для 10 – код 1010, для 8 – код 1000).
```

3.3.8. Умовний вираз

Операція, що вводиться двома лексемами `'?'` і `':'` (ранг 13), є унікальною. По-перше, до неї входить не одна, а дві лексеми, по-друге, вона тримісна, тобто повинна мати три операнда. За її допомогою формується умовний вираз, що має такий вигляд:

```
операнд_1 ? операнд_2 : операнд_3
```

Всі три операнда – вирази. **`Операнд_1`** – це арифметичний вираз і найчастіше – відношення або логічний вираз. Типи **`операнда_2`** і **`операнда_3`** можуть бути різними (але вони повинні бути одного

типу або повинні автоматично приводитися до одного типу).

Перший операнд є умовою, в залежності від якої обчислюється значення виразу в цілому. Якщо значення першого операнда відмінно від нуля (умова істинна), то обчислюється значення **операнда_2**, і воно стає результатом. Якщо значення першого операнда дорівнює 0 (тобто умова помилкова), то обчислюється значення **операнда_3**, і воно стає результатом.

3.4. Функції printf() і scanf()

3.4.1. Використання функцій printf() і scanf()

Функції **printf()** і **scanf()** дозволяють організувати взаємодію з програмою та мають назву функцій вводу-виводу. В мові **C** доступні й інші функції вводу-виводу, але саме **printf()** і **scanf()** є найбільш універсальними. Історично склалося так, що вони, як і усі інші функції в бібліотеці **C**, не були частиною визначення мови. Спочатку мова **C** залишала реалізацію засобів вводу-виводу розробникам компіляторів. Це зробило можливим покращити відповідність функцій вводу-виводу конкретним машинам. В інтересах сумісності різні реалізації постачалися зі своїми версіями функцій **scanf()** і **printf()**. Однак між реалізаціями зустрічалися деякі розбіжності. В **C89** і **C99** описані стандартні версії цих функцій, які і будуть розглядатися далі.

Хоча **printf()** є функцією виводу, а **scanf()** – функцією вводу, обидві вони працюють дуже схожим чином, використовуючи керувальний рядок і список аргументів. Розглянемо по черзі функції **printf()** і **scanf()**.

3.4.2. Функція printf()

Інструкції, які даються функції **printf()** для виводу змінної, залежать від типу цієї змінної. Наприклад, раніше ви вже використовували форму запису %d, для виводу цілого числа, %f – для виводу числа с рухомою комою та %c – для виводу символу. Ці позначення мають назву **специфікаторів**

перетворення, оскільки вони визначають, яким чином дані перетворюються у форму, що відображається на екрані консолі.

Наведемо список специфікаторів перетворення, які стандарт **ANSI C** надає для функції **printf()**, і потім покажемо, як використовувати найбільш загальні з них. В табл. 3.8 наведені специфікатори перетворення та показаний вивід, до якого вони призводять.

Таблиця 3.8 – Специфікатори перетворення

Специфікатор перетворення	Опис виводу
1	2
%a	Число з рухомою комою, шістнадцяткові цифри та **p**-запис (**C99**/**C11**)
%A	Число з рухомою комою, шістнадцяткові цифри та **P**-запис (**C99**/**C11**)
%c	Одиночний символ
%d	Десяткове ціле число зі знаком
%e	Число з рухомою комою, експоненціальне представлення
%E	Число з рухомою комою, експоненціальне представлення
%f	Число з рухомою комою, десяткове представлення
%g	В залежності від значення використовує %f або %e. Специфікатор %e використовується, якщо показник степені менше -4 або більше чи дорівнює вказаній точності
%G	В залежності від значення використовує %f або %E. Специфікатор %E використовується, якщо показник степені менше -4 або більше чи дорівнює вказаній точності
%i	Десяткове ціле число зі знаком (те ж саме, що й %d) Вісімкове ціле число без знаку
%o	Вісімкове ціле
*p	Вказівник
%s	Символьний рядок
%u	Десяткове ціле число без знаку
%x	Шістнадцяткове ціле число без знаку. Використовуються шістнадцяткові цифри 0f
%X	Шістнадцяткове ціле число без знаку. Використовуються шістнадцяткові цифри 0F
%%	Знак процента

Розглянемо програму, в якій використовуються деякі специфікатори перетворення. Текст програми буде мати такий вигляд:

```
    #include <stdio.h>
    #include <windows.h>
    #define  PI 3.141593

int main(void)
{
    int number = 7;
    float pies = 12.75;
    int cost = 7800;

    SetConsoleOutputCP(1251);

   printf("Значення pi дорівнює %f.\n", PI);
   printf("%d%% учасників змагань з\'їли %f пиріжків з вишнями.\n",
       number, pies);
   printf("До побачення! "
          "Ваші   здібності   занадто   дорого   обходяться.\n");
printf("Ми через вас втратили %c%d\n", '$', 2 * cost);

   return 0;
}
```

Результат роботи програми наведено на рис. 3.2.

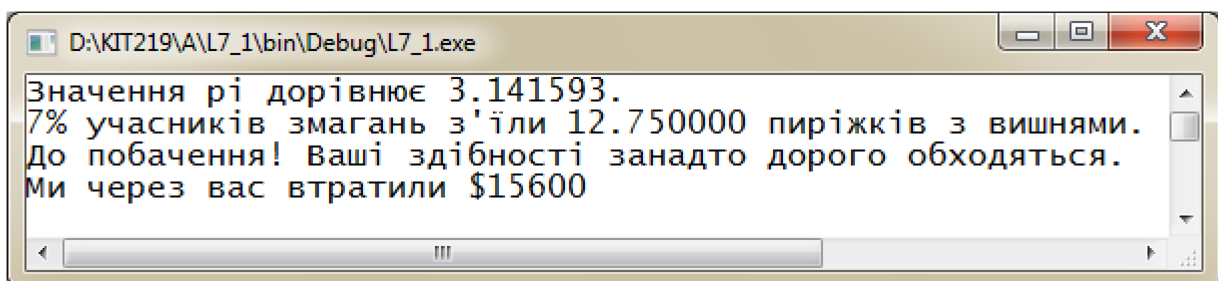


Рисунок 3.2 – Результат використання деяких специфікаторів перетворення

Формат використання функції **`printf()`** має такий вигляд:

```
printf(керувальний_рядок, елемент_1, елемент_2, ...);
```

В цьому форматі `елемент_1`, `елемент_2` і т. д. – це елементи, які треба вивести. Ними можуть бути змінні, константи або навіть вирази, які обчислюються до того, як значення буде виведено. `Керувальний_рядок` являє собою символьний рядок, який описує спосіб

виводу елементів. Він повинен містити специфікатор перетворення для кожного елемента, що підлягає виводу.

Розглянемо оператор

```
printf("%d%% учасників змагань з\'їли %f пиріжків з вишнями.\n",
    number, pies);
```

В цьому операторі `керувальний_рядок` – це фраза, що обмежена подвійними лапками. Він містить два специфікатори перетворення %d і %f, що відповідають `number` і `pies` – двом елементам, які виводяться, а також специфікатор %% для виводу самого знаку %.

Розглянемо ще один рядок програми:

```
printf("Значення pi дорівнює %f.\n", PI);
```

На цей раз список елементів складається тільки з одного елемента – символічної константи **PI**

Рядок програми

```
printf("Ми через вас втратили %c%d\n", '$', 2 * cost);
```

показує приклад, коли у якості елементів `елемент_1` і `елемент_2` функції **printf()** використовується символ `'$'` і вираз `2 * cost`.

Модифікатори специфікаторів перетворення для функції. Базовий специфікатор перетворення **printf()** можна змінювати, вставляючи модифікатори між знаком % і символом, який визначає перетворення.

В табл. 3.9 та 3.10 наведені символи, які можна розміщувати у якості модифікаторів та прапорців.

При зазначенні більш ніж одного модифікатора, вони повинні розташовуватися в тому порядку, в якому представлені в табл. 3.10. Не всі можливі комбінації є припустимими. В таблиці зазначені доповнення стандарту **C99**. Ваша реалізація може не підтримувати усі вказані варіанти.

Таблиця 3.9 – Модифікатори функції **printf()**

Модифікатор	Опис
1	2
прапорець	П'ять допустимих прапорців (**-**, **+**, **пробіл**, **#** і **0**) описані в табл. 1.3. Можна вказувати декілька прапорців або не вказувати їх зовсім. **Приклад:** "%-10d
цифра (цифри)	Мінімальна ширина поля. Якщо число або рядок, які не вміщуються в це поле, буде використовуватися поле більшої ширини. **Приклад:** "%4d"
.цифра (.цифри)	Точність. Для перетворень %e, %E і %f вказується кількість цифр, які будуть виведені справа від десяткової крапки. Для перетворень %g і %G задається максимальна кількість значущих цифр. Для перетворень %s визначається максимальна кількість символів, які можуть бути виведені. Для цілочислових перетворень вказується мінімальна кількість цифр, що відображаються; за необхідності для відповідності з цим мінімумом застосовуються провідні нулі. Використання тільки крапки (.) передбачає, що далі йде нуль, тобто %.f – те ж саме, що й %.0f. **Приклад:** "%5.2f" виводить значення типу **float** у полі шириною п'ять символів і двома цифрами після десяткової крапки
h	Використовується зі специфікатором цілочислового перетворення для відображення значень типів **short int** або **unsigned short int**. **Приклади:** "%hu", "%hx" і "%6.4hd".
hh	Використовується зі специфікатором цілочислового перетворення для відображення значень типів **signed char** або **unsigned char**. **Приклади:** "%hhu", "%hhx" і "%6.4hhd".
j	Використовується зі специфікатором цілочислового перетворення для відображення значень **intmax_t** або **uintmax_t**. Ці типи визначені в файлі **stdint.h**. **Приклади:** "%jd" і "%8jX".
l	Використовується зі специфікатором цілочислового перетворення для відображення значень типів long intабо unsigned long int. **Приклади:** "%ld" і "%8lu".
ll	Використовується зі специфікатором відображення значень типів long long (стандарт C99). **Приклади:** "%lld" і "%8llu".
L	Використовується зі специфікатором перетворення значень з рухомою комою для відображення значень типу long double. **Приклади**: "%Lf" і "%10.4Le".
t	Використовується зі специфікатором цілочислового перетворення для відображення значень ptrdiff_t. Цей тип відповідає різниці між двома вказівниками (стандарт C99). **Приклади:** "%td" і "%12ti".
z	Використовується зі специфікатором цілочислового перетворення для відображення значень size_t. Цей тип повертається операцією sizeof (стандарт C99). **Приклади:** "%zd" і "%12zx".

Таблиця 3.10 – Прапорці функції **printf()**

Прапорець	Опис
-	Елемент вирівнюється вліво, тобто зміст буде виведений, починаючи від лівої межі поля. **Приклад**: "%-20s".
+	Значення зі знаком виводяться зі знаком «+», якщо вони позитивні, та зі знаком «-», якщо вони від'ємні. **Приклад**: "%+6.2f".
пробіл	Значення зі знаком виводяться з провідним пробілом (але без знаку), якщо вони позитивні, та зі знаком «-», якщо вони від'ємні. Прапорець + перевизначає дію пробілу. **Приклад**: "% 6.2f".
#	Використовує альтернативну форму для специфікаторів перетворення. Виводить провідний 0 для форми %o і провідний 0x або 0X для форм %x і %X. Для усіх форм з рухомою комою прапорець # гарантує, що символ десяткової крапки буде виведений, навіть якщо за ним не йдуть цифри. Для форм %g и %G це запобігає видаленню завершальних нулів. **Приклади**: "%#o", "%#8.0f" і "%+#10.3E".
0	Для числових форм цей прапорець призводить до заповнення порожніх позицій поля провідними нулями, а не пробілами. Даний прапорець ігнорується, якщо присутній прапорець - або, якщо для цілочислової форми вказана точність. **Приклади**: "%010d", "%08.3f".

Давайте поглянемо на модифікатори, які щойно були описані, в дії.

Почнемо з оцінки впливу модифікатора, який встановлює ширину поля, на вивід цілого числа. Текст відповідної програми буде мати такий вигляд:

```
#include <stdio.h>
#define  PAGES 959
int main(void)
{
printf("*%d*\n",     PAGES);
printf("*%2d*\n",    PAGES);
printf("*%10d*\n",   PAGES);
printf("*%-10d*\n", PAGES);
return 0;
}
```

Програма виводить задане число чотири рази, використовуючи чотири різних специфікатори перетворення (рис. 3.3). Зірочка (*) слугує для позначення початку та кінця кожного поля.

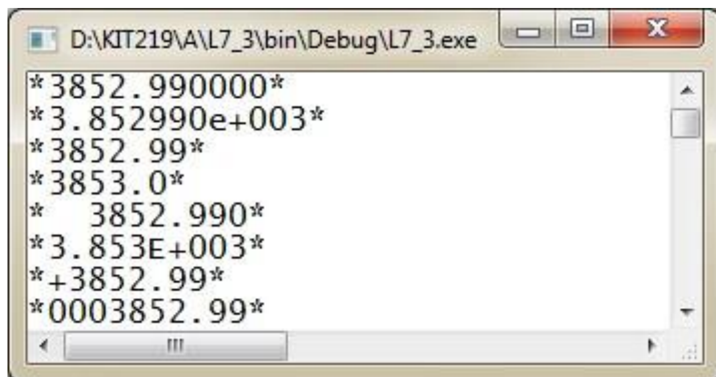


Рисунок 3.3 – Результат використання деяких специфікаторів перетворення

Першим специфікатором перетворення є %d без модифікаторів. Він виробляє поле з шириною, яку має виведене ціле число. Цей варіант прийнятий за замовчуванням, тобто, якщо не надані подальші інструкції, то число буде виведено саме в такому вигляді. Другий специфікатор перетворення – %2d. Він встановлює ширину поля, яка дорівнює 2, але оскільки в даному прикладі ціле число має три значущих цифри, поле автоматично розширюється, щоб вмістити це число. Наступний специфікатор перетворення – %10d. Він генерує поле шириною 10 символів. При цьому в підсумку отримують сім пробілів і три цифри між зірочками, а число зміщено до правої межі поля.

Останнім специфікатором є %-10d. Він також виробляє поле шириною 10 символів, а знак - означає, що число починається з лівого краю, як і було заявлено. Звикнувши до цієї системи, можна переконатися, що вона проста у використанні та забезпечує високий контроль над зовнішнім виглядом виводу.

Тепер розглянемо формати чисел з рухомою комою. Текст програми, яка виводить числа з рухомою комою має такий вигляд:

```
#include <stdio.h>
int main(void) {
    // константа, яка оголошена з ключовим словом const
const double RENT = 3852.99;

printf("*%f*\n",           RENT););
printf("*%e*\n",           RENT);
printf("*%4.2f*\n",        RENT);
printf("*%3.1f*\n",        RENT);
printf("*%10.3f*\n",       RENT);
printf("*%10.3E*\n",       RENT);
printf("*%+4.2f*\n",       RENT);
printf("*%010.2f*\n",      RENT);

return 0;
}
```

Результат роботи програми наведено на рис. 3.4.

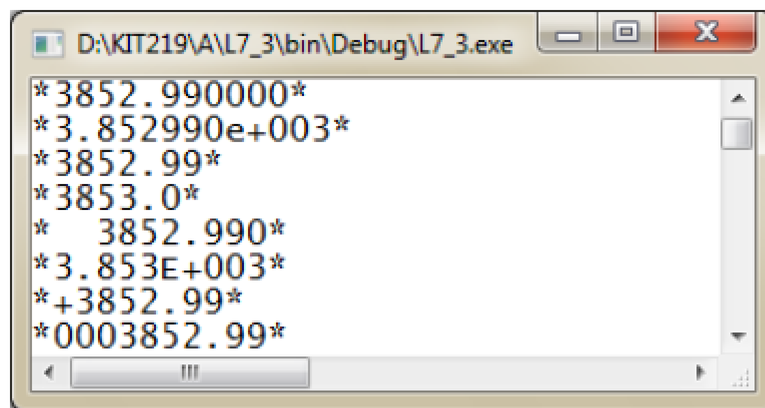


Рисунок 3.4 – Виведення на консоль чисел з рухомою комою

Приклад починається з версії, що застосовується за замовчуванням – %f. В цьому випадку задіяні два стандартних параметра: ширина поля та кількість цифр праворуч від десяткового дробу. Кількість цифр за замовчуванням дорівнює шести, а ширина поля повинна бути такою, щоб вмістити число.

Потім використовується ще одна версія специфікатора, яка прийнята за замовчуванням – %e. Вона виводить одну цифру коду зліва від десяткової крапки і резервує шість позицій праворуч від неї. Виходить досить багато цифр. Щоб виправити це становище, потрібно вказати кількість десяткових позицій праворуч від десяткового дробу, і наступні чотири приклади служать ілюстрацією такого рішення. Зверніть увагу на те, що в четвертому і шостому

прикладі при виведенні відбувається округлення. До того ж у шостому прикладі замість специфікатора e застосовується E.

Нарешті, прапор + призводить до виведення результату з його алгебраїчним знаком, яким в даному випадку є «плюс», а прапор 0 забезпечує доповнення до повної ширини поля провідними нулями. Слід зазначити, що в специфікаторі %010.2f перший 0 – це прапорець, а решта цифри до десяткового дробу (10) вказують ширину поля.

Текст програми, яка демонструє виведення ще кількох можливих комбінацій специфікаторів, має такий вигляд:

```
#include <stdio.h>

int
main(void) {
     printf("%x %X %#x\n", 31, 31, 31);
     printf("**%d**% d**% d**\n", 42, 42, -
     42);
     printf("**%5d**%5.3d**%05d**%05.3d**\n", 6, 6, 6, 6);

     return
0; }
```

Результат виконання програми наведено на рис 3.5.

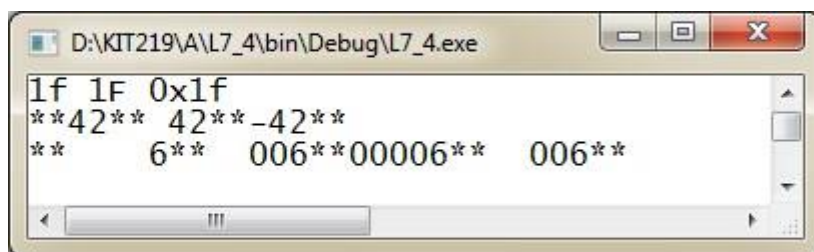


Рисунок 3.5 – Виведення різних комбінацій специфікаторів

В першу чергу відзначимо, що 1f – це шістнадцятковий еквівалент десяткового числа 31. Специфікатор x видає результат 1f, а специфікатор X – 1F. Використання прапорця # забезпечує виведення провідних символів 0x.

Другий рядок виводу ілюструє, що застосування пробілу в специфікаторі призводе до появи провідного пробілу для позитивних, але не для від’ємних значень. Це дозволяє отримати гарний вивід, оскільки позитивні та від’ємні

значення з однаковою кількістю значущих цифр виводяться в полях однакової ширини.

У третьому рядку показано, що використання специфікатора точності (%5.3d) з цілочисловою формою доповнює число провідними нулями до отримання мінімальної кількості цифр (трьох в даному випадку). Однак застосування прапорця 0 призводе до доповнення представлення числа провідними нулями, яких достатньо для заповнення всієї ширини поля. Нарешті, при одночасному застосуванні прапорця 0 і специфікатора точності, прапорець 0 ігнорується.

Напишемо **C**-програму, яка розглядає деякі варіанти виведення рядка символів. Вона буде мати такий вигляд:

```
#include <stdio.h>
#define BLURB "Authentic imitation!"

int main(void)
{
     printf("[%2s]\n",          BLURB);
     printf("[%24s]\n",         BLURB);
     printf("[%24.5s]\n",       BLURB);
     printf("[%-24.5s]\n",      BLURB);

     return 0;
```

Результат виконання програми наведено на рис 3.6.

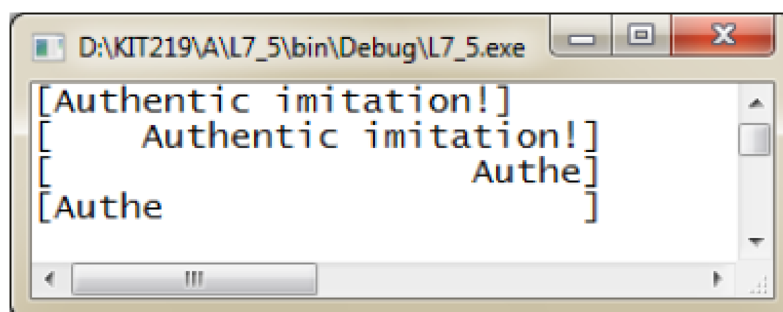


Рисунок 3.6 – Варіанти виведення рядку символів

Зверніть увагу, що специфікатор %2s розширює поле настільки, щоб вмістити всі символи рядку. Крім того, специфікатор точності обмежує кількість символів, що виводяться. Конструкція .5 в специфікаторі формату повідомляє функції **printf()** про те, що треба вивести тільки п’ять символів. Модифікатор -вирівнює текст по лівому краю.

Функція в мові **C** в загальному випадку має значення, що повертається, тобто те, що вона обчислює та повертає в програму, яка її викликала. Наприклад, в бібліотеці **C** є функція **sqrt()**, яка приймає число в якості аргументу та повертає його квадратний корінь. Значення, що повертається, може бути присвоєно змінній, приймати участь в обчисленнях, передаватися як аргумент – словом, їм можна маніпулювати подібно будь-якому іншому значенню. Функція **printf()** також має значення, яке повертається, – кількість символів, що виводиться. Якщо виникла помилка виводу, **printf()** поверне від’ємне значення.

Значення, яке повертається функцією **printf()**, є побічним ефектом її головної задачі з виводу даних і зазвичай не використовується. Єдиною причиною роботи зі значенням, яке повертає функція **printf()**, є необхідність проведення перевірки на предмет наявності помилок виводу. Частіше за все це робиться під час виводу на екран, а під час запису до файлу. Наприклад, якщо запис на **CD-** або **DVD-**диск є неможливим через його переповнення, програма могла б ужити відповідні дії, наприклад, подати звуковий сигнал протягом **30** секунд.

Розглянемо приклад, який демонструє роботу зі значенням, що повертається функцією **printf()**. Текст відповідної програми має такий вигляд:

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

int main(void)
{
     int bph2o = 212;
     int rv;
```

```
     SetConsoleOutputCP(1251);

    rv = printf("Вода закипає при %d градусах за Фаренгейтом.\n",
      bph2o);
    printf("Функція printf() передала на консоль %d символів.\n",
      rv);

    return 0;
}
```

Результат виконання програми наведено на рис 3.7.

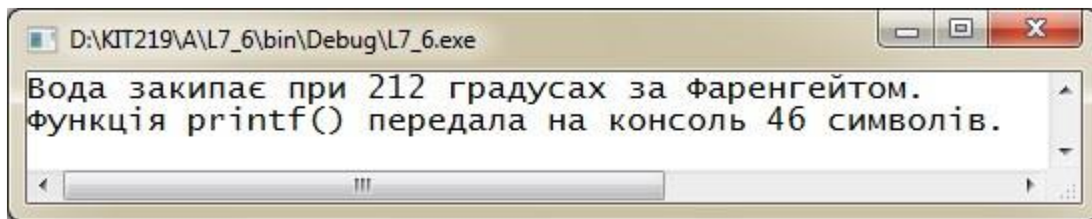


Рисунок 3.7 – Отримання кількості символів, що виводяться функцією **printf()**

Для присвоювання змінній **rv** значення, що повертається функцією **printf()**, в програмі використовується оператор виду `rv = printf(...);`. Він вирішує дві задачі: виводить інформацію на консоль і присвоює значення змінній. Зверніть увагу, що підсумковий результат містить усі символи, які були виведені, в тому числі пробіли та невидимий символ нового рядка.

Іноді оператори **printf()** виявляються надто довгими, щоб уміститися в одному рядку тексту програми. Оскільки в мові **C** пробільні символи (тобто символи пробілу, табуляції та нового рядка) ігноруються в усіх випадках крім ситуації, коли вони використовуються для розділення елементів, оператор можна розмістити в декількох рядках, за умови, що розриви рядків розміщуються строго між елементами.

Розглянемо програму, яка демонструє три можливості розбиття рядка під час виводу на консоль. Текст програми має такий вигляд:

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

int main(void)
{
     SetConsoleOutputCP(1251);
```

```
	printf("Перший спосіб виводу ");
	printf("довгого рядка.\n");
	printf("Другий спосіб виводу \
довгого рядка.\n");
	printf("Третій,   самий   новий   спосіб   виводу   "   "довгого 
рядка.\n");

	return 0;
}
```

Результат виконання програми наведено на рис 3.8.

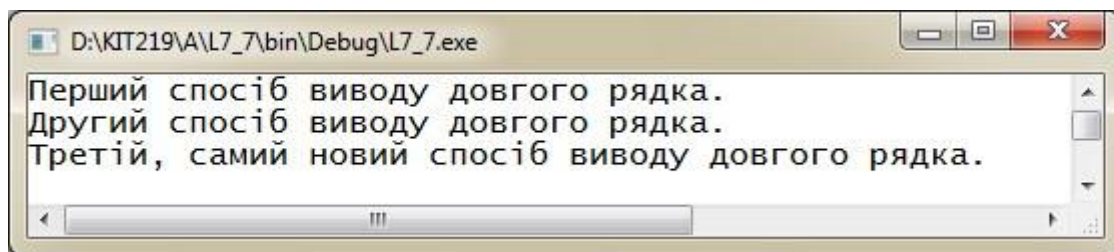


Рисунок 3.8 – Результат роботи програми, яка в своєму тексті застосовує три способи розбиття довгих рядків

Перший метод передбачає застосування більш ніж одного оператора **`printf()`**. Оскільки перший виділений рядок не закінчується символом **`'\n'`**, то другий рядок продовжується з кінця першого.

Другий метод передбачає завершення першого рядка комбінацією зворотної косої риски та натиснення клавіші **`<Enter>`**. Це призводить до того, що текст на екрані починається з нового рядка, але не розміщує символ нового рядка всередині самого рядка. В результаті рядок продовжується на наступному рядку екрану. Однак, як видно з рис. 7.7, наступний рядок екрану повинен починатися з крайньої лівої позиції. Якщо додати відступ довжиною, скажімо, п'ять пробілів, то ці п'ять пробілів стануть частиною рядка.

Третій метод, який був введений у стандарті **`ANSI C`**, має назву «конкатенація рядків». Якщо одна рядкова константа, яка міститься між двома подвійними лапками, йде за іншою рядковою константою, і вони розділені тільки пробільними символами, то ця комбінація трактується мовою як один рядок.

3.4.3. Функція **scanf()**

Тепер перейдемо від виводу до вводу та проведемо дослідження функції **scanf()**. Бібліотека **C** містить декілька функцій вводу, і **scanf()** є найбільш універсальною з них, оскільки має можливість зчитувати дані різних форматів. Зрозуміло, що дані, які вводяться з клавіатури, є текстом, оскільки натиснення клавіш призводить до генерації текстових символів: букв, цифр і розділових знаків. Коли необхідно ввести, наприклад, ціле число 2014, с клавіатури вводяться символи '2', '0', '1' і '4'. Якщо необхідно зберегти його як числове, а не рядкове значення, то програма повинна виконати посимвольне перетворення рядка в числове значення – саме це й робить функція **scanf()**. Вона перетворює рядковий ввід в різні форми: цілі числа, числа з рухомою комою, символи та рядки **C**. Її дія є протилежною до дії функції **printf()**, яка перетворює цілі числа, числа з рухомою комою, символи та рядки **C** в текст, який потім відображається на екрані. Подібно до **printf()**, у функції **scanf()** використовується керувальний рядок, за яким йде список параметрів. Керувальний рядок вказує цільові типи даних для потоку символів, що вводяться. Головна різниця між ними пов'язана зі списком аргументів. У функції **printf()** застосовуються імена змінних, константи та вирази, а у **scanf()** – вказівники на змінні. На щастя, для використання цієї функції знання вказівників не потрібне. Достатньо запам'ятати наступні прості правила:

- якщо функція **scanf()** використовується для того, щоб прочитати значення для змінної одного з базових типів, необхідно перед іменем змінної поставити символ &;

- якщо функція **scanf()** використовується для читання рядка символьного масиву, символ & не потрібен.

Розглянемо програму, яка ілюструє використання функції **scanf()**. Текст програми має такий вигляд:

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

```
#include<windows.h>
int main(void)
{
     int       age;       // змінна
     float   assets;      // змінна
     char    pet[30];     // рядок символів

     SetConsoleCP(1251);
     SetConsoleOutputCP(1251);

     printf("Введіть інформацію про свій вік, "
            "рахунок в банку та улюблену тварину.\n");
     // необхідно вказати символ &
     scanf("%d %f", &age, &assets);
       // для масиву символів & не потрібен
     scanf("%s", pet);
     printf("\n\nВік:                %d\n", age);
     printf("Рахунок у банку:  $%.2f\n", assets);
     printf("Улюблена тварина: %s\n", pet);

     return 0;
}
```

Результат виконання програми наведено на рис. 3.9.

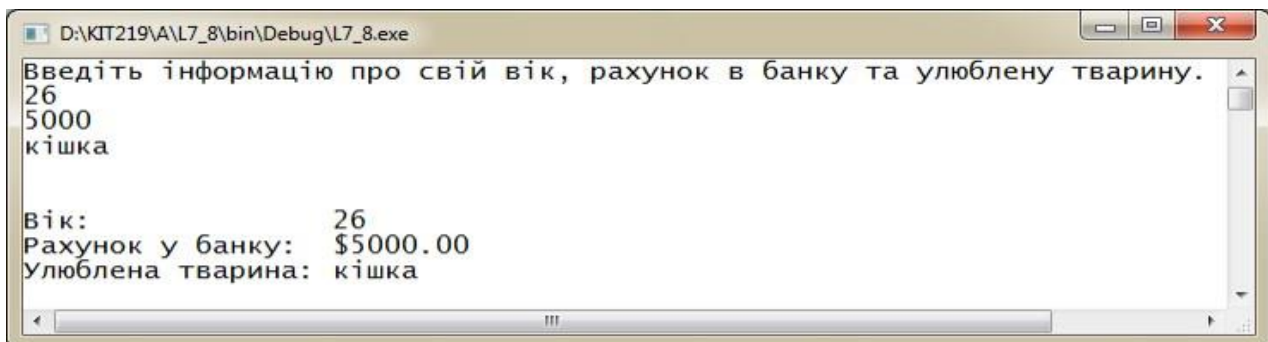


Рисунок 3.9 – Введення даних в одному рядку

При прийнятті рішення про розподіл введених даних на окремі поля, функція **`scanf()`** керується пробільними символами (символами нового рядка, табуляції та пробілу). Вона зіставляє специфікатори перетворення, що йдуть послідовно, з послідовно зазначеними полями, пропускаючи проміжні пробільні символи. Зверніть увагу на дані, що вводяться в один рядок. Дані можна було б вводити і в два, і в три рядки, дотримуючись умови, що між окремими елементами повинен бути, щонайменше, один символ нового рядка, пробілу або табуляції (рис. 3.10).

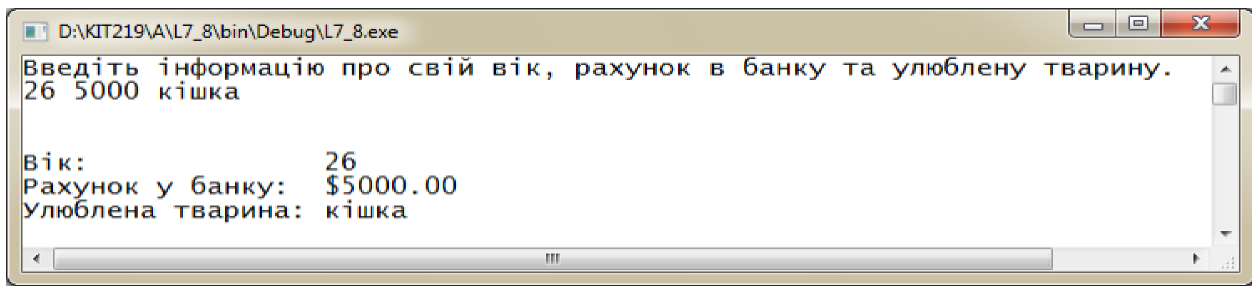


Рисунок 3.10 – Введення даних в три рядки

Єдиним виключенням є специфікатор перетворення %c, який зчитує кожен наступний символ, навіть якщо він є пробільним. У функції **scanf()** використовується в основному той самий набір специфікаторів перетворення, що й у функції **printf()**. Головна відмінність полягає в тому, що у функції **printf()** використовуються специфікатори %f, %e, %E, %g і %G для типів **float** і **double**, тоді як у **scanf()** вони використовуються тільки для типу **float**, вимагаючи позначення модифікатора **l** для типу **double**. В табл. 3.11 наведені основні специфікатори перетворення, як вони описані в стандарті **C99**.

Таблиця 3.11 – Специфікатори перетворення **ANSI C** для функції **scanf()**

Специфікатор перетворення	Значення
%c	Інтерпретує введені дані як символ
%d	Інтерпретує введені дані як десяткове ціле число зі знаком
%e, %f, %g, %a	Інтерпретує введені дані як число з рухомою комою (**%a** з'явився у **C99**)
%E, %F, %G, %A	Інтерпретує введені дані як число з рухомою комою (**%A** з'явився у **C99**)
%i	Інтерпретує введені дані як десяткове ціле число зі знаком
%o	Інтерпретує введені дані як вісімкове ціле число зі знаком
%p	Інтерпретує введені дані як вказівник (адреса)
%s	Інтерпретує введені дані як рядок. Введення починається з першого символу, що не є пробільним, і містить усі символи до наступного пробільного символу
%u	Інтерпретує введені дані як десяткове ціле число без знаку
%x, %X	Інтерпретує введені дані як шістнадцяткове ціле число зі знаком

В специфікаторах перетворення можна також використовувати модифікатори, які зазначені в табл. 7.5. Модифікатори розміщуються між знаком % і буквою перетворення. У випадку застосування в специфікаторі декількох модифікаторів вони повинні з’являтися в тому ж самому порядку, як вони описані в табл. 3.12.

Таблиця 3.12 – Модифікатори перетворення функції **`scanf()`**

Модифікатор	Значення
*	Подавляє присвоювання. **Приклад**: "**%*d**".
цифра (цифри)	Максимальна ширина поля. Введення припиняється, коли досягнута максимальна ширина поля або якщо виявлено перший символ пробілу (в залежності від того, що станеться раніше). **Приклад**: "**%10s**".
hh	Зчитує ціле число як **`signed char`** або **`unsigned char`**. **Приклади**:"**%IiI^**","**%IiИГ**.
ll	Зчитує ціле число як **`long long`** або як **`unsigned long long`** (стандарт **`C99`**). **Приклади**:"**⁰^**", "**%llu**".
h, l або L	Специфікатори "**%hd**" і "**%hi**" вказують, що значення буде збережено з типом **`short int`**. Специфікатори "**%ho**", "**%hx**" і "**%hu**" визначають, що значення буде збережено з типом **`unsigned short int`**. Специфікатори "**%ld**" і "**%li**" вказують, що значення буде збережено з типом **`long`**. Специфікатори "**%lo**", "**%lx**" і "**%lu**" визначають, що значення буде збережено з типом **`unsigned long`**. Специфікатори "**%le**", "**%lf**" і "**%lg**" вказують, що значення буде збережено з типом **`double`**. Використання модифікатора **L** замість **l** у поєднанні з **e**, **f** і **g** визначає, що значення буде збережено з типом **`long double`**. У відсутності цих модифікаторів **d**, **i**, **o** і **x** вказують на тип **`int`**, а **e**, **f** і **g** - на тип **`float`**.
j	Коли за ним йде специфікатор цілочислового перетворення, вказує на використання типів intmax t або uintmax t (стандарт C99). Приклади: "%jd", "%ju".
z	Коли за ним йде специфікатор цілочислового перетворення, вказує на використання типу, що повертається операцією sizeof (стандарт C99). Приклади: "%zd", "%zo".
t	Коли за ним йде специфікатор цілочислового перетворення, вказує на використання типу, який призначений для представлення різниці між двома вказівниками (стандарт C99). Приклади: "%td", "%tx".

Як бачите, застосування специфікаторів перетворення може бути досить різноманітним, причому в таблицях були показані далеко не усі засоби. Ці засоби пов'язані головним чином з полегшенням читання обраних даних з жорстко форматованих джерел, таких як перфокарти або інші записи даних.

Розглянемо, як функція **`scanf()`** зчитує потік даних, що вводяться. Припустимо, що застосовується специфікатор %d, щоб прочитати ціле число. Функція **`scanf()`** починає читати потік вводу по одному символу за один раз. Вона пропускає пробільні символи (символи пробілу, табуляції та нового рядка) до тих пір, поки не натрапить на символ, який не є пробільним. Оскільки функція **`scanf()`** намагається прочитати ціле число, вона очікує виявити цифровий символ або, можливо, знак (**+** або **-**). Зустрівши цифру або знак, вона запам'ятовує цей символ і зчитує наступний. Якщо це цифра, то вона зберігає її та читає наступний символ. Функція **`scanf()`** продовжує читання та збереження символів, поки не натрапить на нецифровий символ. Тоді функція приходить до висновку, що вона досягла кінця чергового цілого числа. Функція **`scanf()`** поміщує цей нецифровий символ назад в потік вводу. Це означає, що наступного разу, коли програма почне зчитування потоку вводу, вона почне його з нецифрового символу, який був відхилений раніше. Нарешті, функція **`scanf()`** обчислює числове значення, що відповідає зчитаним нею цифрам (і, можливо, знаку), і заносить це значення до вказаної змінної.

Якщо використовується ширина поля, функція **`scanf()`** припиняє зчитування коли досягне кінця поля або на першому пробільному символі, в залежності від того, що відбудеться раніше.

А що буде, коли перший відмінний від пробільного символ являє собою, скажімо, символ **A**, а не цифру? В такому випадку функція **`scanf()`** одразу зупиняється й поміщає символ **A** назад в потік вводу. Вказаній змінній значення не присвоюється, і наступного разу, коли програма буде читати потік вводу, вона знову почне його з **A**. Якщо застосовувати в програмі тільки

специфікатори %d, то функція **scanf()** ніколи не просунеться далі цього символу **A**. Крім того, якщо використовувати **scanf()** з декількома специфікаторами, то мова **C** потребує, щоб функція припиняла читання потоку вводу при першій відмові.

Читання потоку вводу з використанням інших числових специфікаторів відбувається таким самим чином, як і у випадку специфікатора %d. Головна різниця між ними полягає в тому, що функція **scanf()** може розрізняти більше символів в якості частини числа. Наприклад, специфікатор %x вимагає, щоб функція **scanf()** розрізняла символи **a-f** і **A-F** як шістнадцяткові цифри. Специфікатори з рухомою комою вимагають, щоб функція **scanf()** розрізняла десяткові крапки, експоненціальну форму запису та нову **p**-нотацію.

Якщо використовувати специфікатор %s, то допускається будь-який символ, відмінний від пробільного, тому функція **scanf()** пропускає пробільні символи до появи першого непробільного символу, після чого зберігає усі непробільні символи аж до наступної появи пробільного символу. Це означає, що специфікатор %s примушує функцію **scanf()** читати одиночне слово, тобто рядок, який не містить пробільних символів. У випадку зазначення ширини поля **scanf()** припиняє читання при досягненні кінця поля або на першому пробільному символі, в залежності від того, що відбудеться раніше. За допомогою ширини поля неможливо примусити функцію **scanf()** читати більше одного слова для одного специфікатора %s. І останній момент: коли функція **scanf()** поміщає рядок до визначеного масиву, вона додає завершальний символ **'\0'** з тим, щоб зробити вміст масиву рядком **C**.

Якщо задати специфікатор %c, то усі символи, що вводяться, запам'ятовуються в початковому вигляді. Якщо наступним символом, що вводиться, є символ пробілу або нового рядка, то він і присвоюється вказаній змінній (пробільні символи не пропускаються).

Функція **scanf()** повертає кількість елементів, які вона успішно прочитала. Якщо не прочитано жодного елемента, як це буває у випадку набору нечислового рядка: в той час коли **scanf()** очікує число, повертається **0**.

При виявленні умови, що має назву «кінець файлу» (**end of file**), функція повертає **EOF** – спеціальне значення, яке визначено у файлі **stdio.h**.

Зазвичай за допомогою директиви **#define** константі **EOF** присвоюється значення **-1**.

3.4.4. Модифікатор * у функціях printf() і scanf()

І в функції **printf()**, і в функції **scanf()** модифікатор ***** можна застосовувати для зміни значення специфікатора, але робиться це по-різному. Для початку розглянемо використання модифікатора ***** у функції **printf()**.

Припустимо, що не треба фіксувати ширину поля заздалегідь, а бажано, щоб її визначила сама програма. Це можна зробити, вказавши замість числа, що задає ширину поля, модифікатор *****. Але знадобиться також додати аргумент для повідомлення функції, якою повинна бути ширина поля. Тобто при наявності специфікатора перетворення %*d список аргументів повинен містити значення для модифікатора ***** і значення для d. Такий метод можна застосовувати також зі значеннями з рухомою комою, щоб вказувати точність і ширину поля.

Розглянемо невеличкий приклад, який демонструє використання модифікатора *****. Текст програми має такий вигляд:

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

int main(void)
{
     unsigned width,
     precision;
     int        number = 256;
     double weight = 242.5;
```

```
        SetConsoleOutputCP(1251);

        printf("Введіть ширину поля: ");
        scanf("%d", &width);
        printf("Значення дорівнює: %*d:\n\n", width, number);
        printf("Тепер введіть ширину та точність: ");
        scanf("%d %d", &width, &precision);
        printf("Bara = %*.*f\n", width, precision, weight);
        printfCAnro^BO !\n");

        return 0;
}
```

Результат виконання програми наведено на рис. 3.11

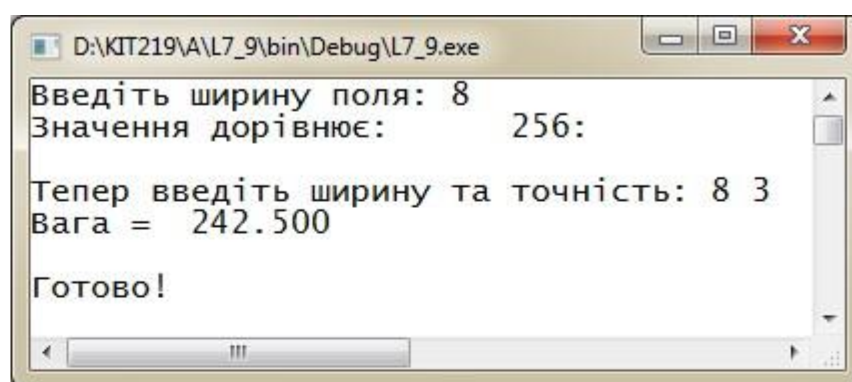


Рисунок 3.11 – Приклад використання модифікатора *****

В цьому прикладі відповіддю на перше питання було число **8**, тому саме **8** використовується для позначення ширини поля. Друга відповідь призвела до встановлення ширини поля, яка дорівнює **8**, і відображенню трьох цифр справа від десяткової крапки. В цілому програма могла б прийняти рішення відносно значень для цих змінних після аналізу значення **weight**.

У випадку функції **scanf()** модифікатор ***** служить зовсім іншій цілі. Коли він розміщений між символом **%** і буквою специфікатора, модифікатор ***** змушує функцію пропускати відповідний ввід.

Розглянемо приклад програми, яка демонструє пропуск вводу певної інформації. Вона має такий вигляд:

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

```
int main(void)
{
      int n;

      SetConsoleOutputCP(1251);

      printf("Введіть три цілих числа: ”);
      scanf(”%*d %*d %d”, &n);
      printf(”\nOcTaHHiM цілим числом було %d\n”, n);

      return 0;
}
```

Оператор **scanf()** вказує програмі на необхідність пропуску двох цілих чисел і копіювання третього цілого числа у змінну n. Така можливість пропуску корисна, якщо програмі, наприклад, треба читати конкретний стовпець у файлі, в якому дані організовані у вигляді уніфікованих стовпців. Результат виконання цієї програми наведено на рис. 3.12.

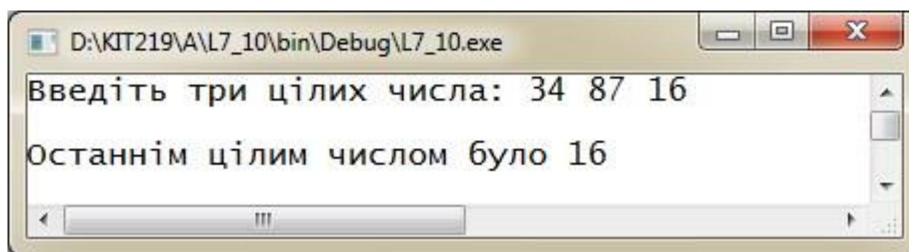


Рисунок 3.12 – Приклад пропуску вводу цілих чисел

Контрольні запитання та завдання

1. В чом полягають основні важливі особливості у мові програмування *С*?
2. Перерахуте переваги мови С.
3. *Які* питання необхідно з’ясувати при вивченні базових понять мови *С*?
4. *Які* базові службові слова визначені у мові програмування *С*?
5. Що відноситься до специфікаторів типів?
6. Що належить до кваліфікаторів типу?
7. *Які* базові типи даних існують у мові програмування *С*?
8. *Які* особливості використання цілочислових типів даних?
9. *Як* у програмі описати змінну з іменем **x** цілого типу?

10. Як у змінну цілого типу записати число?

11. Які особливості мають типи даних з рухомою комою?

12. Як описати змінну, що приймає значення з рухомою комою?

13. Як у змінну з рухомою комою записати числові значення?

14. Які особливості використання даних типу **char** (символьних даних) у програмі на *C*?

15. Які особливості використання даних типу **_Bool** (логічний тип)?

16. Як визначити розмір того чи іншого типу в *C*?

17. Яким чином здійснюється ініціалізація змінних різних типів?

18. Яким чином визначити максимально допустиме (мінімально допустиме) значення змінної певного типу?

19. Які особливості використання типу **enum**?

20. Які особливості застосування типу **void** у програмах на *C*?

21. Чи можна оголошувати змінну типу **void** у програмі?

Завдання для самостійного розв’язання

1. Напишіть невеличку *C*-програму, яка виводить на консоль довільний україномовний текст в чотири рядки. Потім пропустіть один рядок і на наступному вкажіть своє прізвище та ініціали, а також зазначте номер групи. Зробіть копію екрана з результатом роботи програми.

2. Напишіть С-програму, яка б обчислювала об’єм пам’яті, що займає типи данних **`short`** та **`unsigned long int.`** Порівняйте їх між собою з точки зору об’єму пам’яті, яку вони займають.

3. Напишіть С-програму, яка б обчислювала об’єм пам’яті, що займає типи данних **`signed int`** та **`float.`** Порівняйте їх між собою з точки зору об’єму пам’яті, яку вони займають.

4. Напишіть С-програму, яка реалізує роботу з різними специфікаторами та модифікаторами.

5. Напишіть С-програму, яка вводить з клавіатури на окремому рядку дані цілого типу без знаку для числа, місяця та року вашого народження. Виведіть на консоль дату вашого народження у вигляді: ЧЧ.ММ.РРРР. Якщо число або місяць народження складається з однієї цифри, додайте попереду неї нуль програмним шляхом.

4. КЕРУЮЧІ СТРУКТУРИ

4.1. Інструкції вибору в *C*

Зазвичай інструкції в програмі виконуються одна за одною, в тому порядку, в якому вони розміщуються в тексті програми. Це відповідає *послідовном виконанню* програми. Однак у мові **C** є також інструкції, які дозволяють вказувати, яка з інструкцій повинна бути виконана наступною. Цей прийом означає *передачу керування* в іншу точку програми.

Ще у 60-х роках минулого століття стало очевидним, що безконтрольне використання інструкцій передачі керування є коренем певних проблем, які доводиться вирішувати колективам розробників. Винуватицею була визнана інструкція **goto**, яка дозволяла передавати керування практично в будь-яку точку програми. В ті роки новий термін «структурне програмування» став майже синонімом поняття «відмова від **goto**».

У своїх дослідженнях італійські математики Коррадо Бом (Corrado Böhm) і Джузеппе Якопіні (Giuseppe Jacopini) у 1966 році сформулювали теорему, відповідно до якої будь-який алгоритм, що виконується, може бути перетворений до структурованого виду, тобто може бути представлений у вигляді композиції трьох керуючих структур: *структури послідовного виконання, структури вибору* та *структури повторення.*

Іншими словами, вони показали, що будь-яку програму можна написати *без* використання інструкції **goto**. Дійсність вимагала від програмістів прийняти новий стиль «програмування без **goto**». Але по-справжньому новий стиль почав використовуватися тільки у 1970-х, коли до структурованого програмування почали відноситися серйозно. Результати перевершили всі очікування: колективи розробників все частіше робили заяви про зменшення термінів на розробку, все частіше системи почали поставляться вчасно, і все частіше вартість розробки проектів почала відповідати відведеному бюджету. Програми, які розроблялися з використанням прийомів структурного

програмування, виявлялися більш зрозумілими, більш простими у відлагодженні та доробці, а кількість первинних помилок в них стала набагато меншою.

Як вже зазначалося, структура послідовного виконання є надто простою – комп'ютер виконує інструкції одна за одною, в порядку їх розміщення в програмі.

В мові **C** існує три типа *структур вибору, що мають* форму *інструкцій.* Інструкція вибору **if** *обирає* (виконує) дію, якщо умова *істинна,* або пропускає її, якщо умова *хибна.* Інструкція вибору **if...else** виконує одну дію, якщо умова *істинна,* та іншу дію – якщо умова *хибна.* Інструкція вибору **switch** виконує одну з *множини* різних дій в залежності від значення керуючого виразу. Інструкція **if** має назву *інструкції єдиного вибору,* інструкція **if...else** – *інструкція подвійного вибору*, а інструкція **switch** має назву *інструкції множинного вибору,* тому що вона робить вибір з множини дій.

В мові **C** існує також три типа *структур повторення,* що мають форму *інструкцій,* а саме: інструкція **while**, інструкція **do...while** та інструкція **for**.

Таким чином, в **C** є тільки *сім* керуючих структур: структура послідовного виконання, три типи структур вибору та три типи структур повторення. Будь-яка програма на мові **C** є комбінацією керуючих структур цих типів, які відповідають вимогам алгоритмів, реалізованих програмою. Існує також спосіб зв'язування керуючих структур, який оснований на *вкладенні їх одна в одну.*

4.1.1. Інструкція вибору if

Інструкції вибору **if** використовуються для обрання однієї з наявних альтернатив. Наприклад, припустимо, що за 100-бальною системою задовільною оцінкою на екзамені вважається оцінка 60. Інструкція

```c
if(grade >= 60)
{
puts("Здано");
} // кінець оператора if
```

перевіряє істинність умови `grade >= 60`. Якщо умова істинна, за допомогою функції `puts()` виводиться рядок `"Здано"` (пройдено), і виконання продовжується з інструкції, яка йде далі.

Функція `puts()` виводить рядок до стандартного потоку виводу. Після виводу рядка відбувається перехід на новий рядок (вивід символу «новий рядок»). Символ кінця рядку (тобто нульовий символ) не виводиться.

Якщо умова є хибною, вивід рядка пропускається, і виконання продовжується одразу з наступної інструкції. Другий рядок в цій структурі вибору має відступ. Відступи в таких випадках є необов'язковими, але вони допомагають бачити структуру програми. Компілятор **C** ігнорує пробільні символи (пробіли, символи табуляції та переведення рядка), які використовуються для оформлення відступів і відокремлення логічних блоків пустими рядками.

Блок-схема на рис. 4.1 ілюструє інструкцію єдиного вибору `if`.

Рисунок 4.1 – Передача керування для інструкції єдиного вибору `if`

Логічний блок у формі ромбу містить вираз, який відіграє роль умови, що може мати або істинне значення, або хибне. З логічного блока виходять дві стрілки. Одна відповідає істинному значенню умовного виразу в блоці, а друга – хибному. Рішення можуть прийматися з застосуванням операторів відношень або перевірки рівності. В дійсності рішення можуть прийматися на основі будь-яких виразів: якщо вираз повертає нульове значення, воно інтерпретується як хибне, будь-яке ненульове значення інтерпретується як істинне.

4.1.2. Інструкція вибору if...else

Інструкція **if...else** дозволяє вказати дві різні дії: одна виконується, коли умова істинна, друга – коли умова хибна. Наприклад, інструкція

```
if(grade >= 60)
{
   puts("Здано");
} // кінець if  else
{
   puts("Не здано");
} // кінець else
```

виведе `"Здано"`, якщо студент отримав оцінку **60** або вищу, і `"Не здано"`, якщо оцінка нижче **60**. В будь-якому випадку, після виводу того або іншого рядка, виконання буде продовжено з інструкції, яка йде наступною. Тіло гілки **else** також має відступ. Блок-схема на рис. 4.2 ілюструє, як відбувається виконання цієї інструкції **if...else**.

4.1.3. Умовний оператор та умовний вираз

В мові **C** є *умовний оператор* **(?:)**, який дуже нагадує *інструкцію* **if...else**.

Умовний оператор – це єдиний **тернарний (тримісний) оператор** в мові **C**. Він приймає три операнда, і разом з умовним оператором вони утворюють **умовний вираз**.

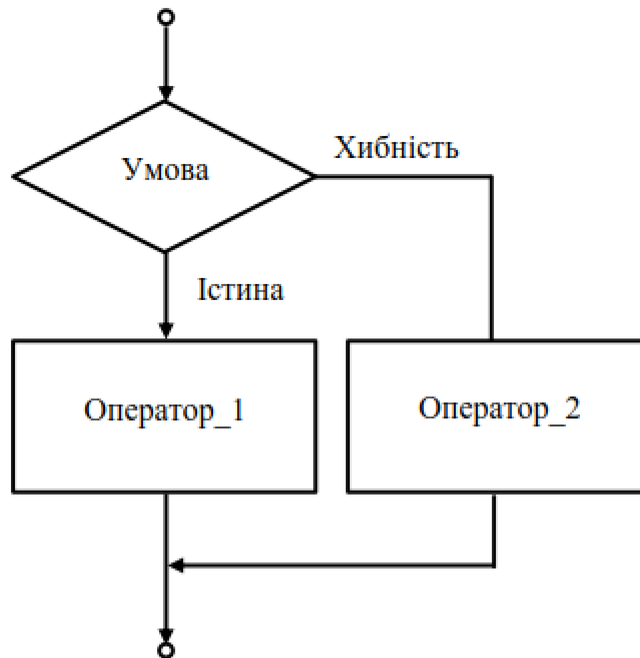


Рисунок 4.2 – Передача керування для інструкції подвійного вибору **if...else**

Перший операнд – це умова.

Другий операнд – значення всього умовного виразу, коли умова істинна. *Третій операнд* – значення всього умовного виразу, коли умова хибна. Наприклад, функція **puts()**

```
puts(grade >= 60 ? "Здано" : "Не здано");
```

отримує в якості аргументу умовний вираз, який повертає рядок "Здано", якщо умова **grade >= 60** істинна, і рядок "Не здано", якщо умова хибна. Функція **puts()**, по суті, виконує те ж саме, що й попередня інструкція **if...else**.

Другий і третій операнди в умовному виразі можуть бути також діями. Наприклад, умовний вираз

```
grade >= 60 ? puts("Здано") : puts("Не здано");
```

читається так: «Якщо оцінка **grade** більше або дорівнює **60**, то виконати **puts("Здано")**, в протилежному випадку – **puts("Не**

здано")». Цю інструкцію ще можна порівняти з попередньою інструкцією **if...else**. Існують ситуації, коли умовний вираз використовувати можна, а інструкцію **if...else** – ні.

4.1.4. Вкладені інструкції if...else

Вкладені інструкції **if...else** дозволяють перевірити декілька умов шляхом вкладення інструкцій **if...else** одна в одну. Наприклад,

```
if (вираз) оператор_1;
else if (вираз) оператор_2;
     else оператор_3;
```

Блок-схема на рис. 4.3 ілюструє, яким чином відбувається виконання вкладених інструкцій **if...else**.

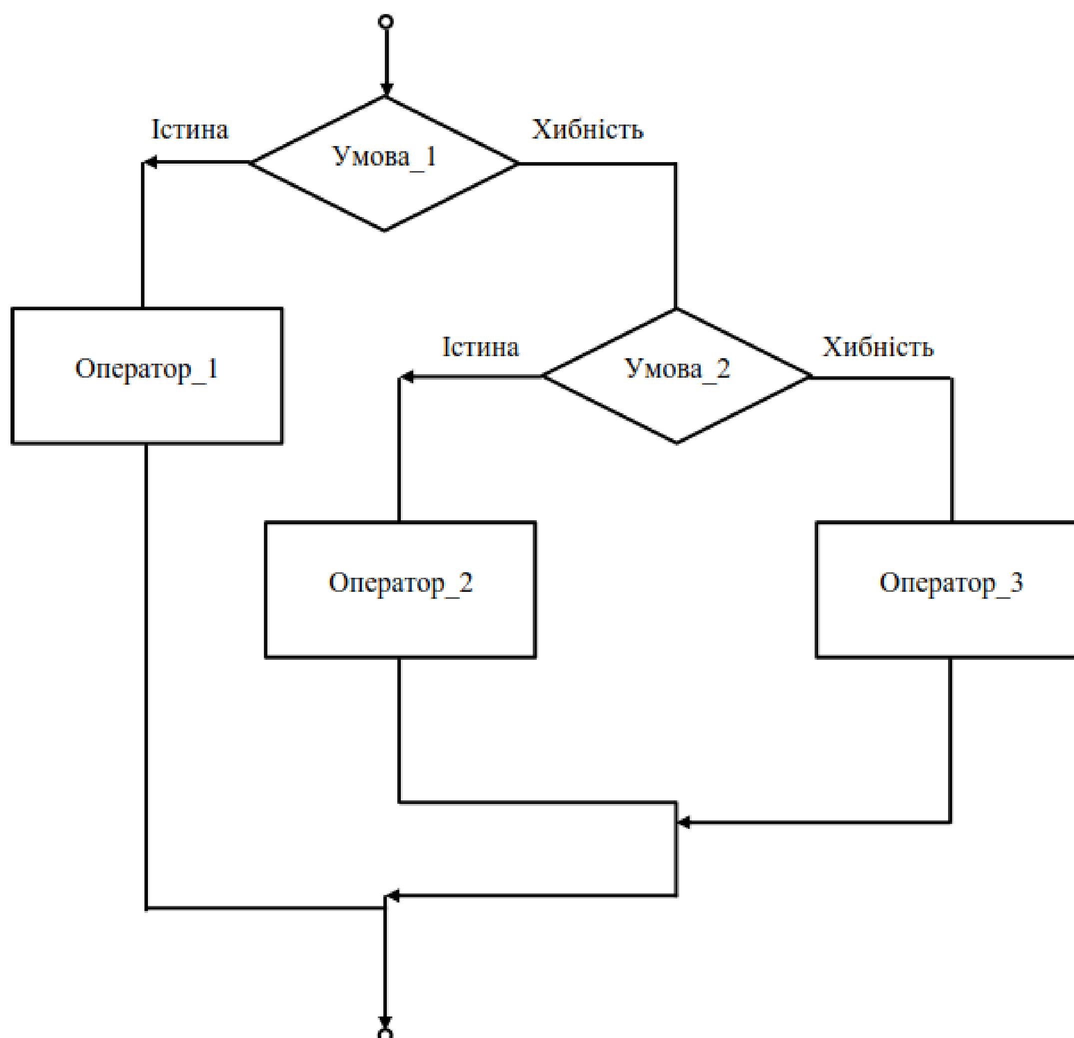


Рисунок 4.3 – Передача керування для вкладених інструкцій **if...else**

Наступний код виведе "Відмінно 5A", якщо студент отримав на екзамені оцінку **90** або вище, "Добре 4B" – якщо оцінка вище або дорівнює **82** (але нижче **90**), "Добре 4C" – якщо оцінка вище або дорівнює **75** (але нижче **82**), "Задовільно 3D" – якщо оцінка вище або дорівнює **64** (але нижче **75**), "Задовільно 3E" – якщо оцінка вище або дорівнює **60** (але нижче 64), і "Незадовільно 2F або FX" – в усіх інших випадках.

```
if(grade  >=90)     //  if_1
{
     puts("Відмінно 5A");
}                   //кінець  if_1
else                //  else 1
{
     if(grade  >=82)     // if_2
     {
          puts("Добре 4B");
     }                   // кінецьif_2
     else                // else_2
     {
          if(grade >= 75)     // if_3
          {
               puts("Добре 4C");
          }                   // кінецьif_3
          else                // else_3
          {
               if(grade >= 64)     // if_4
               {
                    puts("Задовільно 3D");
               }                   // кінець  if_4
               else                // else_4
               {
                    if(grade >= 60)     // if_5
                    {
                         puts("Задовільно 3E");
                    }                   // кінець if_5
                    else                // else_5
                    {
                         puts("Незадовільно 2F або FX"); }
                                        // кінець else_5
               }                   // кінець  else_4
          }                   // кінець else_3
     }                   // кінець else_2
}                   // кінець else_1
```

Якщо значення змінної **grade** більше або дорівнює **90**, то усі п’ять умов будуть істинними, але виконана буде тільки перша інструкція – функція **puts()**. В цьому випадку гілка **else** самої зовнішньої інструкції **if...else** пропускається, і виконання продовжується з першої інструкції, яка йде наступною за цим фрагментом коду.

Попередню інструкцію **if** можна записати інакше:

```
if(grade >= 90)
{
      puts("Відмінно 5A");
} // кінець if
else if(grade >= 82)
{
      puts("Добре 4B");
} // кінець else if
else if(grade >= 75)
{
      puts("Добре 4C");
} // кінець else if
else if(grade >= 64)
{
      puts("Задовільно 3D");
} // кінець else if
else if(grade >= 60)
{
      puts("Задовільно 3E");
} // кінець else_if
else {
      puts("Незадовільно 2F або FX");
} // кінець else
```

З точки зору компілятора **C** обидві форми є еквівалентними. Однак остання користується більшою популярністю, тому що вона позбавляє від необхідності збільшувати відступи від лівого краю. Через великі відступи в рядку може залишитися надто мало місця, що буде вимагати розбиття інструкції на декілька рядків і погіршувати читабельність програми.

Інструкція вибору **if** дозволяє вказати в тілі лише одну інструкцію, тому, якщо тіло інструкції **if** складається з єдиної інструкції, фігурні дужки можна опустити. Щоб включити в тіло **if** декілька інструкцій, їх слід розміщувати у фігурні дужки (**{** і **}**). Множина інструкцій, обмежена парою

фігурних дужок, має назву **складеної інструкції** або **блоку**.

Наступний фрагмент містить складену інструкцію у гілці `else` інструкції `if...else`.

```
if(grade >= 60)
{
      puts("Здано.");
} // кінець if
else {
      puts("Не здано.");
      puts("Ви повинні пройти цей курс знову.");
} // кінець else
```

В даному випадку, якщо оцінка менш ніж **60**, програма виконає обидві функції `puts()` в тілі гілки `else` і виведе

```
Не здано.
Ви повинні пройти цей курс знову.
```

Фігурні дужки, що обмежують дві інструкції у гілці `else`, грають важливу роль. Без них функція `puts("Ви повинні пройти цей курс знову.");` виявилася б за межами тіла гілки `else` і виконувалася б завжди, незалежно від оцінки за екзамен.

На місце єдиної інструкції не тільки можна підставити складену інструкцію, можна також взагалі убрати цю інструкцію, тобто вставити порожню інструкцію. Роль порожньої інструкції відіграє крапка з комою `(;)`.

4.2. Інструкція множинного вибору switch

Іноді алгоритм може передбачати порівняння значення змінної або результату виразу з множиною варіантів і передбачати для кожного з них свою дію. Така ситуація має назву **інструкції множинного вибору**. Формат інструкції множинного вибору має такий вигляд:

```
switch(вираз)
{
      case константний_вираз_1: інструкції;
                                 break;
      case константний_вираз_2: інструкції;
```

```
                          break;
                            ...
case константний_вираз_n: інструкції;
                          break;
default:                      інструкції;
                          break;
```

Інструкція **switch** складається з послідовності міток **case**, необов'язкового варіанту **default** та інструкцій, які виконуються для кожного варіанту.

Кожна гілка **case** помічена однією або декількома цілочисловими константами або ж константними виразами. Обчислення починаються з тієї гілки **case**, в якій константа співпадає зі значенням виразу. Константи усіх гілок **case** повинні відрізнятися одна від одної. Якщо з'ясується, що жодна з констант не підходить, то виконується гілка, яка помічена словом **default** (якщо така мається), в іншому випадку нічого не робиться. Гілки **case** і **default** можна розміщувати в будь-якому порядку.

Інструкція **break** призводить до миттєвого виходу з інструкції **switch**. Оскільки вибір гілки **case** реалізується як перехід на мітку, то після виконання однієї гілки **case**, якщо нічого не застосовувати, програма почне виконувати наступну гілку. Інструкції **break** і **return** – найбільш поширені засоби виходу з інструкції **switch**. Інструкція **break** застосовуються також для примусового виходу з циклів **while**, **for** і **do...while**.

Послідовний прохід по гілкам – річ ненадійна, це може бути пов'язано з помилками, особливо під час модифікації програми. За виключенням випадку з декількома мітками для одного обчислення. Намагайтеся, за можливістю, не користуватися наскрізним проходом, але якщо ви його застосовуєте, обов'язково коментуйте ці особливі місця. Навіть наприкінці останньої гілки (після **default**) розміщуйте інструкцію **break**, хоча з точки зору логіки в ній немає жодної необхідності. Але ця маленька обережність врятує, коли одного разу вам необхідно буде додати в кінець ще одну гілку **case**.

Слід знати про три важливі моменти щодо інструкції **switch**:

1) Інструкція **switch** відрізняється від інструкції **if** тим, що вона може виконувати тільки операції перевірки строгої рівності, в той час як **if** може обчислювати логічні вирази та відношення.

2) Не може бути двох констант в одному операторі **switch**, які мають однакові значення. Звичайно, оператор **switch**, який містить у собі інший оператор **switch**, може мати аналогічні константи.

3) Якщо в операторі **switch** використовуються символьні константи, вони автоматично перетворюються на цілочислові значення.

Інструкція **switch** часто використовується для обробки команд з клавіатури, наприклад, при визначенні яка клавіша була натиснута.

Розглянемо програму для визначення назви дня тижня за його номером, яка використовує інструкцію **switch**. Її текст має такий вигляд:

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

     int main(void) {
         unsigned day;

         SetConsoleOutputCP(1251);

         printf("Який сьогодні день тижня: ");
         scanf("%d", &day);

         if (day > 7) day = day % 7;

         printf("Сьогодні - "); switch(day)
         {
               case 0: printf("Неділя\n");
                       break;
               case 1: printf("Понеділок\n");
                       break;
               case 2: printf("Вівторок\n");
                       break;
               case 3: printf("Середа\n");
                       break;
               case 4: printf("Четвер\n");
                       break;
               case 5: printf("П'ятниця\n");
                       break;
              default: printf("Субота\n");
                       break;
         }
```

```
return 0; }
```

Результат виконання програми наведено на рис. 4.4.

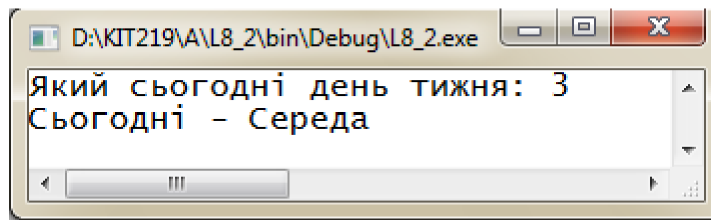


Рисунок 4.4 – Приклад використання інструкції **switch**

4.3. Різновиди циклів в мові С

Цикл – це різновид керівної конструкції у мові **C**, яка призначена для організації багаторазового виконання набору інструкцій (команд).

Також циклом може називатися будь-яка послідовність команд, яка багатократно виконується та організована будь-яким чином (наприклад, за допомогою умовного переходу).

Тіло циклу – послідовність інструкцій, яка призначена для багаторазового виконання.

Ітерація – одноразове виконання тіла циклу.

Умова виходу з циклу або **умова завершення циклу** – вираз, який визначає чергове виконання ітерації або завершення циклу.

Лічильник циклу – змінна, в якій зберігається номер поточної ітерації. Цикл не обов’язково повинен містити лічильник. Також лічильників може бути декілька. Умова виходу з циклу може залежати від декількох змінних, а може визначатися зовнішніми умовами (наприклад, настанням певного часу). В останньому випадку лічильник взагалі не потрібен.

Частинами виконання будь-якого циклу є початкова ініціалізація змінних циклу, перевірка умови виходу з циклу, виконання тіла циклу й оновлення змінної циклу на кожній ітерації.

Крім того, мова **C** надає засоби для керування ходом виконання циклу, наприклад, оператори завершення циклу, тобто оператори виходу з циклу незалежно від істинності умови виходу (інструкція **break**) і оператори

пропущення ітерації (інструкція **continue**).

4.3.1. Цикл з передумовою while

Цикл з передумовою виконується до тих пір, поки умова, що позначена в круглих дужках після ключового слова **while**, є істинною. Ця умова перевіряється до початку виконання тіла циклу, тому тіло може не виконатися жодного разу (якщо умова одразу є хибною). Оскільки у мові **C** цей тип циклу реалізується за допомогою ключового слова **while**, існує інша його назва – цикл **while**.

Структуру циклу **while** умовно можна записати наступним чином:

```
while(умова)
{
// тіло циклу
}
```

Зверніть увагу на те, що наприкінці першого рядка **while**(умова) **крапка з комою не ставиться. Це найпоширеніша помилка, що призводить до зациклювання програми.**

Розглянемо застосування циклу **while** на прикладі руху автомобіля. На псевдокоді це можна записати наступним чином:

```
1) Якщо швидкість автомобіля менше 60 кілометрів на годину
2) Продовжувати збільшувати швидкість на 10 км/год.
```

Істинною умовою циклу в даному випадку є швидкість автомобіля менш ніж 60 км/год, а хибною – швидкість автомобіля, яка більше або дорівнює 60 км/год. Повторення циклу буде тривати до тих пір поки швидкість автомобіля не стане більше або дорівнювати 60 км/год. Після настання цього моменту умова циклу стане хибною, і програма вийде з циклу.

Розглянемо фрагмент коду **C**-програми з застосуванням циклу **while**, що вирішує поставлене завдання:

```
int speed = 5;            // початкова швидкість автомобіля
while(speed < 60)          // заголовок циклу while
    speed += 10;           // тіло циклу
```

Спочатку була оголошена та ініціалізована змінна **speed**. Далі програма перевіряє умову циклу **while** (5 < 60 == **true**). Програма входить до циклу і виконує оператор (speed += 10;). Тепер вже speed = 15. Знову виконується перевірка (15 < 60 == **true**). Умова знову є істинною, і значення змінної змінюється на 25. Таким чином, виконуються послідовні повторення циклу. Змінна **speed** ще приймає значення: 35, 45, 55. Остання зміна призводить до того, що умова в циклі **while** (65 < 60 == **false**) стає помилковою. Відбувається вихід з циклу **while**. Таким чином, тіло циклу виконалося 6 разів.

Текст програми, який виконує застосування циклу **while**, має такий вигляд:

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

int main(void)
{
   int speed;           //  Початкова швидкість автомобіля
   int count;           //  Кількість ітерацій циклу

   SetConsoleOutputCP(1251);

   printf("==============================================\n");
   printf("    Програма   для   демонстрації   циклу   while\n");
   printf("==============================================\n");

   count = 1; speed = 5;
   printf("\n  Початок циклу while\n\n");
   while( speed < 60 )
   {
     printf("Ітерація №%d. Швидкість автомобіля - %2d км/год\n",
     count, speed);
     speed += 10;       // додаємо швидкість
     count++;          // збільшуємо кількість ітерацій на 1
   }
   printf("\n  Закінчення циклу while\n\n");
   return 0;
}
```

Результат роботи програми, яка демонструє застосування циклу **while** наведено на рис. 4.5.

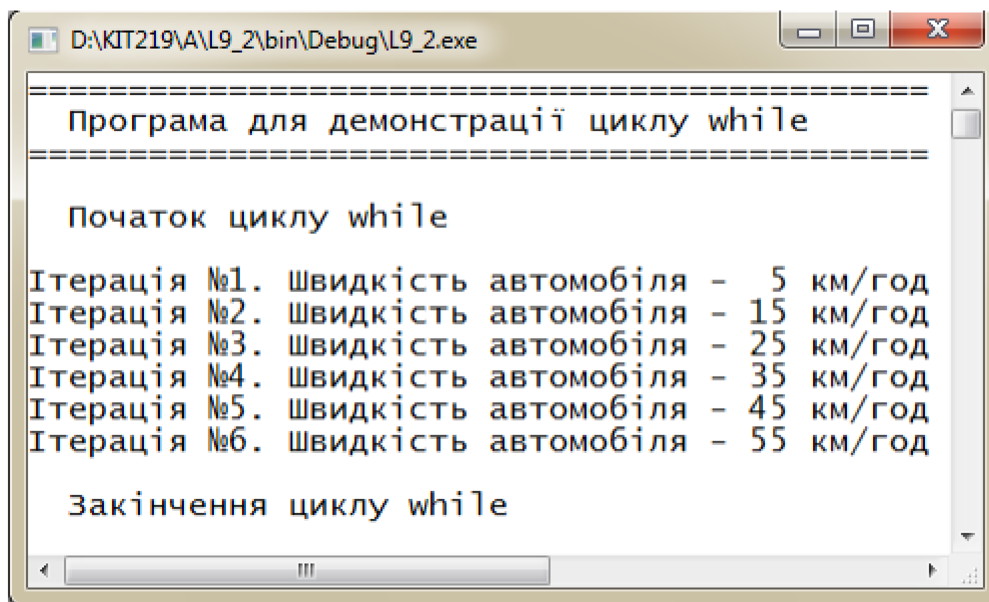


Рисунок 4.5 – Результат роботи програми з циклом **while**

4.3.2. Цикл з постумовою do...while

Цикл з постумовою – це цикл, в якому умова перевіряється після виконання тіла циклу. Тобто тіло циклу завжди виконується хоча б один раз. У мові **C** роль циклу з постумовою відіграє цикл **do...while**.

Структуру циклу **do...while** умовно можна записати наступним чином:

```
do
{
      // тіло циклу
}
while(умова);
```

На початку циклу **do...while** пишеться зарезервоване слово **do**. Потім йдуть фігурні дужки, в яких міститься тіло циклу. Їх можна опускати, в разі використання одного оператора в тілі циклу. Для циклу з постумовою істинна умова в круглих дужках (після **while**) трактується як умова продовження (цикл завершується, коли умова хибна). **Зверніть увагу на те, що наприкінці останнього рядка** **while**`(умова);` **крапка з комою ставиться**.

Розглянемо програму з циклом **do...while**, яка виконує деякі транзакції з грошовим рахунком у комерційному банку.

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

int main(void)
{
    int balance;          // поточний баланс
    int removal;           // зняття грошей з рахунку

    SetConsoleOutputCP(1251);
    printf("//===============================================\n");
    printf("//  Програма для демонстрації циклу do...while  \n");
    printf("//===============================================\n");
    printf("Поточний час:          %s\n\n", __TIME__);
    printf("Поповнити рахунок: ");
    scanf("%d", &balance);
    printf("Рахунок поповнено!\n");
    printf("===============================================\n");
    printf("Ваш поточний рахунок складає:   %2d грн\n", balance);
    printf("===============================================\n");
    printf("\nПочаток циклу do...while\n");
    do   // початок циклу do...while
    {
          printf("\n=========================================\n");
          printf("Поточний баланс:   %d грн\n", balance);
          printf("=========================================\n");
          printf("Зняти з рахунку:    ");
          scanf("%d", &removal);
          printf("=========================================\n");
          balance -= removal;           // знімаємо з рахунку
    } while( balance > 0 ); // кінець циклу do...while
    printf("\n\nКінець циклу do...while\n\n");
    printf("===============================================\n");
    printf("На Вашому рахунку не залишилося коштів!\n");
    printf("===============================================\n");
    return 0;
}
```

Результат роботи програми, яка демонструє застосування циклу **do...while**, наведено на рис. 4.6.

4.3.3. Цикл з лічильником for

Цикл з лічильником – це цикл, в якому змінна циклу модифікує своє значення від заданого початкового значення до кінцевого значення з певним кроком, і для кожного значення цієї змінної тіло циклу виконується один раз.

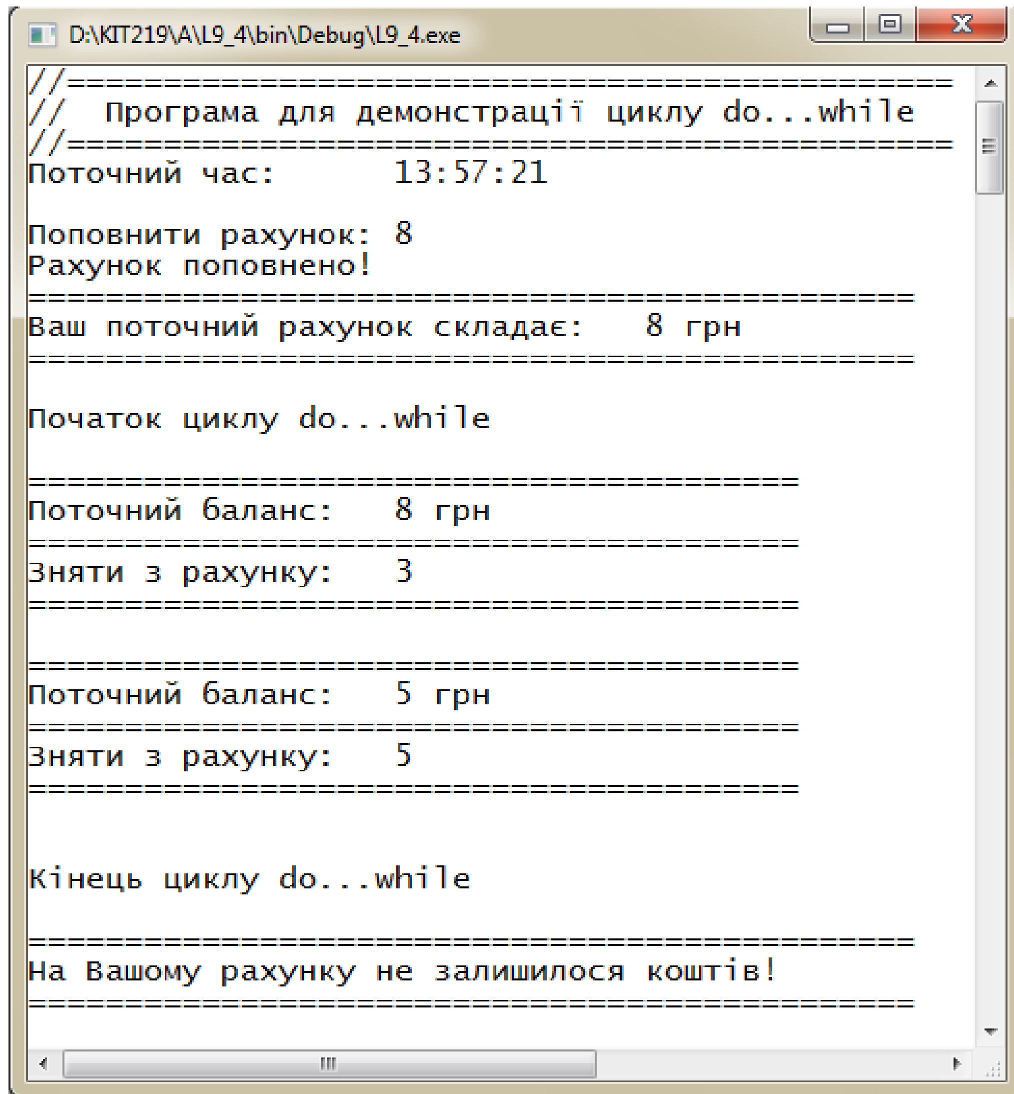


Рисунок 4.6 – Результат роботи програми з циклом **`do...while`**

В мові ***C*** цей тип циклу реалізується оператором **`for`**, в якому вказується лічильник (так звана «змінна циклу»), потрібна кількість проходів (або граничне значення лічильника) і, можливо, крок, з яким змінюється лічильник.

Структуру циклу **`for`** умовно можна записати наступним чином:

```
for(вираз_1; вираз_2; вираз_3)
{
     // тіло циклу
}
```

З точки зору граматики три компоненти циклу **`for`** являють собою

довільні вирази, але частіше за все `вираз_1` – це присвоювання початкового значення, `вираз_3` – збільшення або зменшення кроку циклу, а `вираз_2` – умова виходу з циклу (умовний вираз). **Будь-який з цих виразів може бути відсутнім, але крапку з комою необхідно залишати.** При відсутності компонентів `вираз_1` або `вираз_3` вважається, що їх просто не існує в конструкції циклу. При відсутності компоненту `вираз_2` вважається, що його значення завжди є істинним.

В мові **C** цикл **for**, незважаючи на синтаксичну форму, цикл з лічильником насправді є циклом з передумовою. Тобто в **C** конструкція циклу:

```
for(i = 0; i < 10; i++)
{
     // тіло циклу
}
```

фактично являє собою інший варіант запису конструкції:

```
i = 0;  while(i < 10)
{
     // тіло циклу
     i++;
}
```

Тобто в конструкції **for** спочатку пишеться довільне значення для ініціалізації змінної циклу, потім – умова продовження і, наприкінці циклу, деяка операція, що виконується після кожного тіла циклу.

Розглянемо приклад застосування циклу **for**. Текст програми для виводу на екран послідовності чисел від **0** до **9** буде мати такий вигляд:

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


int main(void)
{
   int i;

   SetConsoleOutputCP(1251);
   printf("//=======================================\n");
   printf("//   Програма  для  демонстрації  циклу  for   \n");
   printf("//=======================================\n");
   printf("\nПочаток циклу for\n\n");
   for(i = 0; i < 10; i++)
```

```
    {
       printf("%3d\n", i); }
    printf("\nЗакінчення циклу for\n\n");
    return 0;
 }
```

Результат роботи програми, яка демонструє застосування циклу **for**, наведено на рис. 4.7.

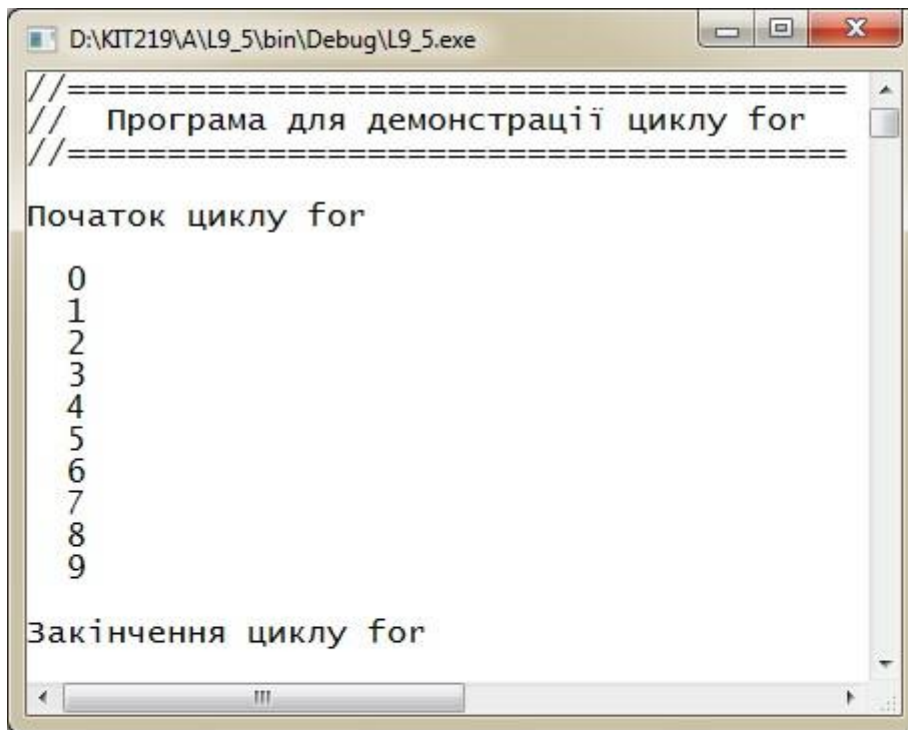


Рисунок 4.7 – Результат роботи програми з циклом **for**

Цикл буде виконуватися до тих пір, поки змінна **i** буде менше **10**. При цьому, після кожної ітерації вона буде збільшуватися на **1** завдяки **i++**.

Стандарт **C99** дозволяє оголошувати змінну у виразі ініціалізації інструкції **for**. Для використання в якості лічильника циклу можна створити нову змінну, оголосивши її в заголовку інструкції **for**, при цьому область видимості такої змінної буде обмежена інструкцією **for**.

Наведемо приклад програми, яка демонструє таку можливість. Текст програми має такий вигляд:

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

```c
int main(void)
{
    SetConsoleOutputCP(1251);

    printf("Значення змінної x\n\n");
    for(int x = 1; x <= 5; ++x)
    {
        printf("%3d", x);
    }
    printf("\n\n");
    return 0;
}
```

Результат виконання програми, яка демонструє нові можливості циклу **for**, наведено на рис. 4.8.

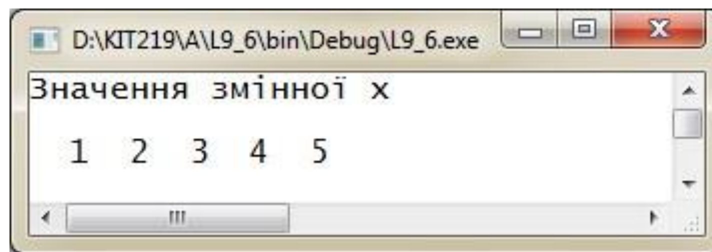


Рисунок 4.8 – Результат роботи програми з циклом **for** з можливістю оголошення змінної в заголовку інструкції

Будь-яка змінна, яка оголошена у заголовку інструкції **for**, буде доступною лише в цій інструкції – до такої змінної неможна буде звертатися за межами інструкції **for**, бо це призведе до помилки компіляції.

4.3.4. Цикл з виходом з середини

Цикл з виходом з середини – це найзагальніший тип умовного циклу. Синтаксично такий цикл оформлюється за допомогою трьох інструкцій: початок циклу, кінець циклу та інструкція (команда) виходу з циклу. Інструкція початку позначає точку програми, з якої починається тіло циклу, інструкція кінця – точку, де тіло закінчується. Всередині тіла має бути присутня команда виходу з циклу, при виконанні якої цикл завершується, і керування передається на оператор, який йде після інструкції кінця циклу. Команда виходу має викликатися не безумовно, а тільки при виконанні умови виходу з циклу.

Принциповою відмінністю такого різновиду циклу від розглянутих вище є те, що частина тіла циклу, яка розташована після початку циклу і до команди виходу, виконується завжди (навіть якщо умова виходу з циклу є істинною при першій ітерації), а частина тіла циклу, яка йде після команди виходу, не виконується під час останньої ітерації. Легко побачити, що за допомогою циклу з виходом з середини легко утворити як цикл з передумовою (розташувавши команду виходу на початку тіла циклу), так і цикл з постумовою (розташувавши команду виходу наприкінці тіла циклу).

Команда дострокового виходу застосовується, коли необхідно перервати виконання циклу, в якому умови виходу ще не досягнуто. Таке буває, наприклад, коли під час виконання циклу зустрічається помилка, після якої подальше виконання циклу не має сенсу.

У мові **C** цикл з виходом з середини може бути побудований за допомогою будь-якого умовного циклу та інструкції дострокового виходу з циклу (такої, як **break** або **exit**) або інструкції безумовного переходу **goto**.

Команда безумовного переходу **goto** здійснює перехід на команду, яка розміщується безпосередньо за межами циклу, всередині якого вона знаходиться. Так у мові **C** два наступних цикли працюють цілком однаково:

```
// Застосування оператора break
while(умова)
{
     // оператори
     if(помилка) break;
     // оператори
}
// продовження програми

// Аналогічний фрагмент без оператора break
while(умова)
{
     // оператори
     if(помилка) goto label;
     // оператори
}
label: // продовження програми
```

В обох випадках, якщо в тілі циклу виконується умова, яка пов'язана з помилкою, буде виконано перехід на оператори, що позначені як «продовження програми». Таким чином, оператор дострокового виходу з циклу, по суті, просто маскує безумовний перехід, однак використанню **break** слід надати перевагу перед використанням **goto**, оскільки:

- поведінка інструкції **break** чітко задана мовою;

- вона є потенційно менш небезпечною (немає, наприклад, імовірності помилитися з розташуванням або назвою мітки переходу);

- явний достроковий вихід з циклу не порушує основ структурного програмування.

4.3.5. Нескінченний цикл

Іноді в програмах використовуються цикли, вихід з яких не передбачений логікою програми. Такі цикли називаються безумовними або нескінченними. Особливих синтаксичних засобів для створення таких циклів, через їхню нетиповість, мова **C** не передбачає. Тому такі цикли створюються за допомогою конструкцій, які призначені для створення звичайних (або умовних) циклів. Для забезпечення нескінченного повторення, перевірка умови в такому циклі відсутня або замінюється константним значенням.

Вихід з таких циклів частіше за все відбувається завдяки використанню будь-якої умови, істинне значення якої дозволяє застосувати інструкцію **break** або **exit**

Розглянемо приклад використання нескінченного циклу, який у першому випадку організований за допомогою циклу **for**, а в другому – за допомогою циклу **while**.

Текст першої програми має такий вигляд:

```
#include <stdio.h>
#include<windows.h>
int main(void)
{
   int   i;     // номер поточного символу
   char  ch;     // введений символ

   SetConsoleOutputCP(1251);
   printf("==============================================\n");
   printf("  Організація нескінченого циклу for( ; ; )    \n");
   printf("==============================================\n");

   i = 1;
   printf("==============================================\n");
   printf("Нескінчений цикл розпочато                     \n");
   printf("==============================================\n\n");
   for( ; ; )
   {
        ch = getchar();            // введення символу
        if(ch == '#')
        {
            printf("Ітерація не завершена\n");
            break;                 // вихід з циклу
        }
        ch = getchar();            // зчитуємо <Enter>
        i++;
        printf("Ітерація завершена\n");
   }
   printf("\n==============================================\n");
   printf("Нескінчений цикл закінчено                     \n");
   printf("==============================================\n\n");
   printf("==============================================\n");
   printf("Введено ознаку виходу з нескінченного циклу #\n");
   printf("Кількість введених символів:  %d\n", i);
   printf("==============================================\n\n");
   return 0;
}
```

Результат роботи програми наведено на рис. 4.9. Умовою виходу з нескінченого циклу є введення символу '#'

```
D:\KIT219\A\L9_7\bin\Debug\L9_7.exe
===============================================
  Організація нескінченого циклу for( ; ; )
===============================================
===============================================
Нескінчений цикл розпочато
===============================================

d
Ітерація завершена
g
Ітерація завершена
q
Ітерація завершена
e
Ітерація завершена
#
Ітерація не завершена

===============================================
Нескінчений цикл закінчено
===============================================

===============================================
Введено ознаку виходу з нескінченного циклу #
Кількість введених символів:   5
===============================================
```

Рисунок 4.9 – Результат роботи програми нескінченого циклу **for(; ;)**

Текст другої програми має такий вигляд:

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

int main(void)
{
   char ch;                          // введений символ

   SetConsoleOutputCP(1251);
   printf("==============================================\n");
   printf("  Організація нескінченого циклу while(1)      \n");
   printf("==============================================\n");
   printf("  Можна натискати клавіші '1', '2' і '#'       \n");
   printf("==============================================\n");
   printf("==============================================\n");
   printf("Нескінчений цикл розпочато\n");
   printf("==============================================\n\n");
   while(1)
   {
        ch = getchar();              // введення символу
        switch(ch)
        {
```

```
            case '1': printf("Натиснута клавіша '1'\n");
                      break;
            case '2': printf("Натиснута клавіша '2'\n");
                      break;
            case '#': printf("Програма завершена\n");
                      exit(0);
             default: printf("Натиснута не та клавіша\n");
                      break;
        }
        ch = getchar();              // зчитуємо <Enter> }
   printf("\n============================================\n");
   printf("Нескінчений цикл закінчено                      \n");
   printf("============================================\n\n");
   return 0;
}
```

Результат роботи програми наведено на рис. 4.10. Умовою виходу з нескінченого циклу є введення символу '#'.

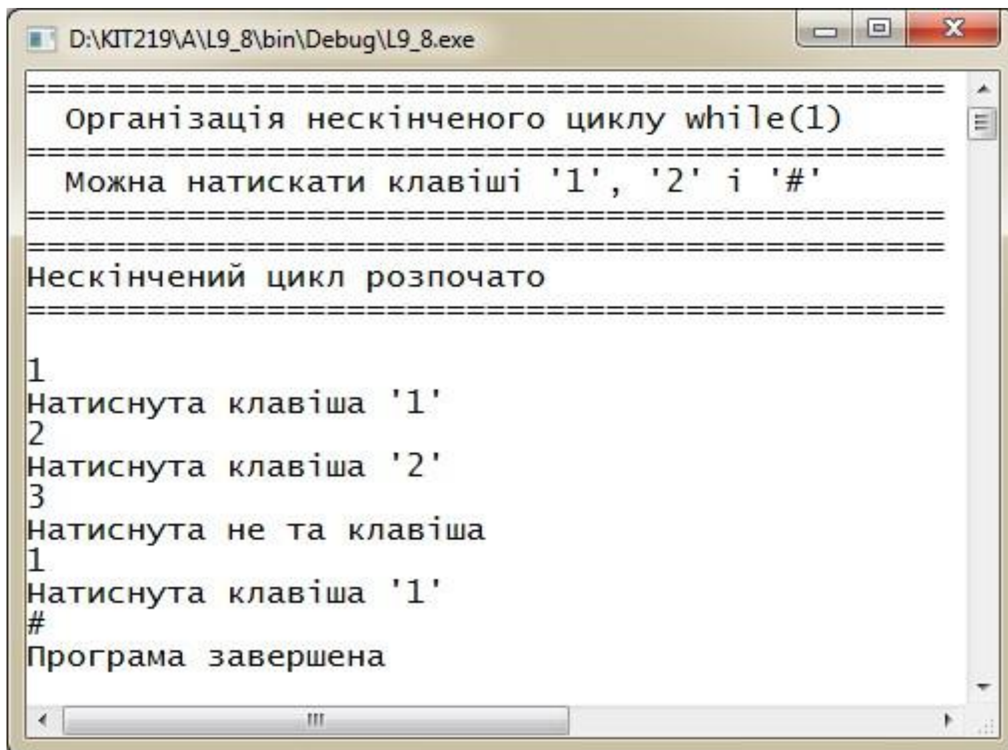


Рисунок 4.10 – Результат роботи програми нескінченого циклу **`while(1)`**

4.4. Пропуск ітерації continue

У мові **`C`** в якості команди пропуску ітерації використовується інструкція **`continue`** в конструкції циклу. Даний оператор застосовується, коли в поточній ітерації циклу необхідно пропустити всі команди до кінця тіла циклу. При цьому сам цикл не переривається, умови продовження або

виходу обчислюються звичайним чином. Дія цього оператора аналогічна безумовному переходу на рядок всередині тіла циклу, наступний за останньою його командою.

Розглянемо приклад застосування інструкції **continue**. Текст програми має такий вигляд:

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

int main(void)
{
    int i = 0, j = 10;

    SetConsoleOutputCP(1251);

    for(int k = 0; k < 20; k++)
    {
       if(k % 5 != 0) continue;
       i++;
       j--;
    printf("k = %2d, i = %2d, j = %2d\n", k, i, j);
    }
    printf("\n");
    return 0;
}
```

Результат виконання програми наведено на рис. 4.11.

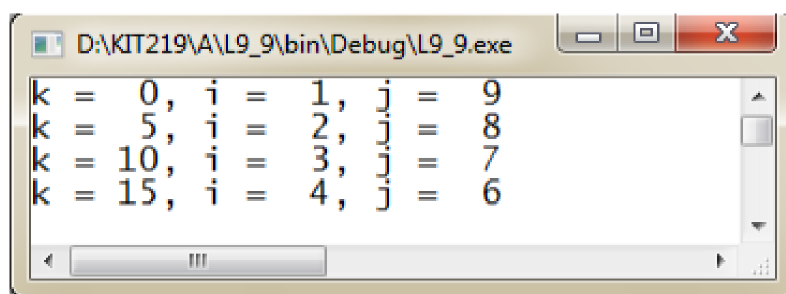


Рисунок 4.11 – Приклад використання інструкції **continue**

4.5. Необхідність використання дострокового виходу з циклу та пропуску ітерації

З погляду структурного програмування команди дострокового виходу з циклу та продовження ітерації є надлишковими, оскільки їх дія може бути легко змодельована чисто структурними засобами. Більш того, на думку

деяких теоретиків програмування (зокрема, Дейкстри), сам факт використання в програмі неструктурних засобів, будь то **goto**, **break** або **continue**, є свідченням недостатньо опрацьованого алгоритму розв'язання задачі.

Однак на практиці код програми часто є записом наявного, раніше сформульованого алгоритму, перепрацьовувати який недоцільно з чисто технічних причин. Спроба замінити в такому коді команду дострокового виходу на структурні конструкції часто виявляється неефективною або громіздкою.

Незважаючи на обмежену корисність і можливість заміни на інші мовні конструкції, команди пропуску ітерації і, особливо, дострокового виходу з циклу в окремих випадках виявляються вкрай корисними, саме через це вони зберігаються в мові **C**.

4.6. Вкладені цикли

В **C** існує можливість утворювати цикл всередині тіла іншого циклу. Такий цикл має назву **вкладеного циклу**. Вкладений цикл, по відношенню до циклу в тіло якого він вкладений, буде йменуватися **внутрішнім циклом**, і навпаки цикл, в тілі якого існує вкладений цикл, буде мати назву **зовнішнього циклу**. Всередині вкладеного циклу може бути наступний вкладений цикл, утворюючи наступний рівень вкладеності і так далі. Кількість рівнів вкладеності, як правило, не обмежується.

Повна кількість виконання тіла внутрішнього циклу не перевищує добутку кількості ітерацій внутрішнього і всіх зовнішніх циклів. Наприклад, взявши три вкладених один в одного цикли, кожний по **10** ітерацій, отримаємо **10** виконань тіла зовнішнього циклу, **100** – для циклу другого рівня і **1000** – в найбільш вкладеному циклі.

Розглянемо приклад використання вкладених циклів **for**. Текст програми має такий вигляд:

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

```
#include<windows.h>
int main(void)
{
   int i, j;

   SetConsoleOutputCP(1251);

   printf("Таблиця множення\n");
   printf("==================================================\n");
   printf(" x |    1    2     3    4    5    6    7    8    9
   \n");
   printf("==================================================\n");

   for(i = 1; i < 10; i++)
   {
        printf("%2d |", i);
        for(j = 1; j < 10; j++)
        {
            printf("%5d", i * j);
        }
        printf("\n"); }
   printf("==================================================\n");
   return 0;
}
```

Результат роботи програми наведено на рис. 4.12.

```
D:\KIT219\A\L9_1\bin\Debug\L9_1.exe
Таблиця множення
==================================================
 x |    1    2    3    4    5    6    7    8    9
==================================================
 1 |    1    2    3    4    5    6    7    8    9
 2 |    2    4    6    8   10   12   14   16   18
 3 |    3    6    9   12   15   18   21   24   27
 4 |    4    8   12   16   20   24   28   32   36
 5 |    5   10   15   20   25   30   35   40   45
 6 |    6   12   18   24   30   36   42   48   54
 7 |    7   14   21   28   35   42   49   56   63
 8 |    8   16   24   32   40   48   56   64   72
 9 |    9   18   27   36   45   54   63   72   81
==================================================
```

Рисунок 4.12 – Приклад використання вкладених циклів **for**

Однією з проблем, що пов’язані з вкладеними циклами, є організація дострокового виходу з них. В мові **C** інструкція **break** забезпечує вихід лише з циклу того рівня, звідки вона викликається. При цьому зовнішній цикл продовжить своє виконання. Проблема може здатися надуманою, але вона дійсно іноді виникає при програмуванні складних алгоритмів обробки

даних, коли алгоритм вимагає негайного переривання за певних умов, наявність яких можна перевірити тільки в глибоко вкладеному циклі.

Розв'язання проблеми виходу з вкладених циклів має декілька варіантів. Найпростіший з них – це використання оператору безумовного переходу

goto для виходу в точку програми, яка йде безпосередньо за вкладеним циклом. Цей варіант критикується прихильниками структурного програмування, як і всі конструкції, що вимагають використання goto.

Альтернатива – використання штатних засобів завершення циклів, які передбачають в разі необхідності встановлення особливих прапорців для негайного завершення обробки. Недолік – ускладнення коду, зниження продуктивності без будь-яких переваг, окрім теоретичної «правильності» через відмову від goto.

Контрольні запитання та завдання

1. Що вирозумієте під терміном «структурне програмування» ?
2. Чим відрізняється умовний вираз від умовної інструкції?
3. Який вигляд має умовний оператор?
4. Що таке повна та неповна форми умовної інструкції?
5. Вирази якого типу можуть визначати умови?
6. Які значення виразу, що визначають умову, вважаються істинними, а які хибними?
7. Які операції відносяться до операцій відношення?
8. Чим відрізняється операція "==" від операції "="?
9. Які операції відносяться до логічних? Який вони мають пріоритет?
10. Чому може дорівнювати значення виразу відношення або логічного виразу?
11. Коли застосовується вкладення умовних інструкцій?
12. Що собою являє інструкція switch? Як нею користуватися?
13. Як записати інструкцію switch за допомогою умовних інструкцій?

14. Перілічіть різновиди циклів в мові *C* .

15. За допомогою якого ключового слова реалызуэться цикл з передумовою?

16. Як називається цикл, в якому умова перевіряється після виконання тіла циклу?

17. Скільки разів та як називається цикл в чкому для кожного значення цієї змінної тіло циклу виконується один раз?

18. Який цил є найзагальнішим типом умовного циклу?

19. Яка інструкція циклу використовується в мові С в якості команди пропуску ітерації?

20. Чому дорівнює повна кількість виконання тіла внутрішнього циклу?

Завдання для самостійного розв'язання

1.Напишіть текст *C*-програми, яка б вводила два операнди **a** і **b** та один з двох можливих символів арифметичної операції, і в залежності від цієї операції обчислювала б значення виразу. Для обчислення виразу необхідно застосувати умовний оператор (**?:**). Наведіть текст *C*-програми та результати її роботи для обох операцій.

2. Напишіть С-програму для обчислення значення функції. В програмі необхідно застосувати вкладену інструкцію **`if...else`** і навести текст програми та копії екранів для усіх трьох гілок алгоритму. Побудувати схему алгоритму роботи прога наводити не треба.

3. Напишіть С-програму у якій здійснюється вивод на екран послідовності чисел від 0 до 9 із викорстанням циклу **`for`**.

4. Наведіть приклад використання нескінченного циклу, який у першому випадку організований за допомогою циклу **`for`**, а в другому – за допомогою циклу **`while`**.

5. Напишіть *C*-програму в якій би застосовалась інструкція **`continue`**.

5. ОРГАНІЗАЦІЯ ОДНОТИПНИХ ДАНИХ

5.1. Масиви

5.1.1. Масиви в мові *C*

При вирішенні задач з великою кількістю даних однакового типу використання змінних з різними іменами, які не упорядковані за адресами пам’яті, ускладнює програмування. В подібних випадках в мові `C` використовують об’єкти, які звуться масивами.

Розглянемо основні поняття, які стосуються масивів.

Масив – це неперервний фрагмент пам’яті, який містить послідовність об’єктів однакового типу та позначається одним іменем.

Елемент масиву (значення елемента масиву) – значення, яке зберігається у визначеній комірці пам’яті, розташованій в межах масиву, а також адреса цієї комірки пам’яті.

Кожен елемент масиву характеризується трьома величинами:

- адресою елемента – адресою початкової комірки пам’яті, в якій розташований цей елемент;

- індексом елемента (порядковим номером елемента в масиві); - значенням елемента.

Адреса масиву – адреса початкового елемента масиву.

Ім’я масиву – ідентифікатор, який використовується для звернення до елементів масиву.

Розмір масиву – кількість елементів масиву.

Розмір елемента масиву – кількість байтів, які займає один елемент масиву.

Масиви є важливим інструментом у багатьох програмах, оскільки дозволяють у зручній формі зберігати множину елементів зв’язаної інформації.

Масив – це сукупність значень однакового типу, наприклад, **`10`** значень типу **`char`** або **`15`** значень типу **`int`**, які зберігаються в пам’яті послідовно.

Масив в цілому має своє ім’я, а доступ до його окремих елементів здійснюється з використанням цілочислового індексу. Наприклад, оголошення

```
float mas[20];
```

повідомляє про те, що **mas** є масивом, який складається з **20** елементів, кожен з яких може містити у собі значення типу **float**. Першим елементом масиву буде **mas[0]**, другим – **mas[1]** і т. д. аж до **mas[19]**.

Зверніть увагу на те, що **нумерація елементів масиву починається з 0**, а не з **1**. Кожному елементу масиву може бути присвоєно значення типу **float**. Наприклад, можна записати такий код:

```
mas[5] = 32.54;
mas[6] = 1.2e+21;
```

По суті, елемент масиву можна використовувати таким самим чином, як це б робилося для змінної того ж самого типу. Наприклад, можна прочитати значення та помістити його в конкретний елемент масиву:

```
scanf("%f", &mas[4]); // читання значення у 5-й елемент масиву
```

Потенційна пастка полягає в тому, що в інтересах швидкості обчислювань коректність вказаного індексу не перевіряється. Нижче наведені приклади помилкових операторів:

```
mas[20] = 88.32; // такий елемент масиву не існує
mas[33] = 828.12; // такий елемент масиву не існує
```

Слід зазначити, що компілятор не виявляє помилки подібного роду. Під час виконання програми ці оператори занесли б дані в комірки пам’яті, які можливо зайняті іншими даними, потенційно спотворюючи вихідні дані програми, або навіть призвели б до її аварійного завершення.

Масив може стосуватися будь-якого типу даних:

```
int nannies[22]; // масив для 22 цілих чисел
char actors[26]; // масив для 26 символів
long big[500]; // масив для 500 цілих чисел типу long
```

Раніше вже обговорювалися рядки, які являють собою спеціальний випадок того, що можна зберігати в масиві типу **char**. Вміст масиву **char** формує рядок, якщо масив містить нульовий символ (**\0**), який позначає кінець рядку (рис. 5.1).

Масив символів, який не утворює рядок

М	а	с	и	в		с	и	м	в	о	л	і	в	.

Масив символів, який утворює рядок за рахунок символу '\0'

М	а	с	и	в		с	и	м	в	о	л	і	в	.	\0

▲ нульовий символ

Рисунок 5.1 – Приклади символьних масивів, перший з яких не утворює рядок, а другий – утворює

Числа, які застосовуються для ідентифікації елементів масиву, звуться індексами або зміщеннями. Індекси повинні бути цілими числами, до того ж, як було вказано раніше, індексація починається з **0**. Елементи масиву зберігаються в пам'яті поруч один з одним (рис. 5.2).

`int mas1[4]     (4 байта на значення типу int)`

2980	45	4728	5
mas1[0]	mas1[1]	mas1[2]	mas1[3]

`char mas2[4]     (1 байт на значення типу char)`

'л'	'і'	'т'	'о'
mas2[0]	mas2[1]	mas2[2]	mas2[3]

Рисунок 5.2 – Масиви mas1(тип **int**) і **mas2** (тип **char**) в пам'яті комп'ютера

5.1.2. Використання циклів for з масивами

Масиви застосовуються досить часто. В наступній програмі зчитуються **10** результатів гри в гольф для подальшої обробки. За рахунок використання масиву немає необхідності оголошувати **10** змінних з різними іменами (по одній для кожного результату). Крім того, для читання вхідних даних можна застосовувати цикл **for**. Програма призначена для підрахунку загальної суми результатів, їх середнього значення та гандикапу, який являє собою різницю між середнім і стандартним результатом, або паром. Пар – це термін в гольфі, що означає кількість ударів по м'ячу, яку необхідно зробити досвідченому гравцю для проведення м'яча в лунку або проходження всіх лунок.

Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#define SIZE 10
#define PAR 72

int main(void)
{
    int index, score[SIZE];
    int sum = 0;
    float average;

    SetConsoleOutputCP(1251);

    printf("Введіть %d результатів гри в гольф:\n", SIZE);
    // Зчитування 10 результатів гри в гольф
    for(index = 0; index < SIZE; index++)
        scanf("%d", &score[index]);
    printf("Введені наступні результати:\n");
    // Перевірка введеної інформації
    for(index = 0; index < SIZE; index++)
        printf("%5d", score[index]);
    printf("\n");
    for(index = 0; index < SIZE; index++)
        sum += score[index];            // додавання результатів
    average = (float) sum / SIZE;       // перевірений метод
    printf("\nСума результатів = %d\n", sum);
    printf("Середнє значення = %.2f\n", average);
    printf("Отриманий гандикап дорівнює %.0f.\n\n", average - PAR);
    return 0;
}
```

Результат виконання програми наведено на рис. 5.3.

```
D:\KIT219\A\L10_1\bin\Debug\L10_1.exe
Введіть 10 результатів гри в гольф:
99 95 109 105 100
96 98 93 99 97 98
Введені наступні результати:
   99   95  109  105  100   96   98   93   99   97

Сума результатів = 991
Середнє значення = 99.10
Отриманий гандикап дорівнює 27.
```

Рисунок 5.3 – Використання циклів **for** для роботи з масивами

Зверніть увагу на те, що хоча у прикладі було набрано **11** чисел, прочиталися тільки **10** з них, тому що цикл читання зчитує тільки **10** значень. Оскільки функція **scanf()** пропускає пробільні символи, можна вводити в одному рядку усі **10** чисел, вводити кожне число в окремому рядку або, як у розглянутому випадку, для розділення значень скористатися сумішшю символівнового рядка та пробілу. Через буферизацію вводу числа передаються до програми тільки після натиснення клавіші **<Enter>**.

Працювати з масивами та циклами значно зручніше, ніж застосовувати **10** операторів **scanf()** і **10** операторів **printf()** для читання та виводу **10** результатів. Цикл **for** пропонує простий спосіб використання індексів масиву. В розглянутому прикладі проілюстровано декілька аспектів, що стосуються стилю програмування.

По-перше, застосування директиви **#define** для створення символічної константи **SIZE**, яка вказує розмір масиву, є дуже зручною ідеєю. Ця константа використовується у визначенні масиву та при встановленні максимального значення індексу в циклах. Якщо пізніше треба буде розширити програму для обробки **20** результатів, достатньо просто перевизначити константу **SIZE**, зробивши її такою, що дорівнює **20**. Вам не доведеться змінювати кожну частину програми, в якій застосовується розмір масиву.

По-друге, конструкція

```
for(index = 0; index < SIZE; index++)
```

є досить зручною для обробки масиву з розміром **SIZE**. Дуже важливо вказувати правильні межі масиву. Перший елемент має індекс **0**, і цикл починається з встановлення значення **index** в **0**. Оскільки нумерація починається з **0**, індексом останнього елемента є **SIZE-1**. Тобто десятий елемент масиву – це **score[9]**. Умова перевірки **index < SIZE** забезпечує це. Останнім значенням змінної **index**, яке буде застосоване, – це **SIZE-1**.

По-третє, в програмах рекомендується виводити на екран значення, які були щойно прочитані. Це допомагає переконатися в тому, що програма обробляє саме ті дані, які очікуються.

По-четверте, зверніть увагу на використання в програмі трьох окремих циклів **for**. Вас може зацікавити питання, чи можна об’єднати деякі з операцій в один цикл? Так, програма стала б компактнішою, однак це суперечило б принципу модульності. Ідея, що лежить в основі цього терміну, полягає в тому, що програма повинна бути розбита на окремі модулі, і кожен модуль повинен вирішувати одну задачу. Це полегшує читання програми. Але що навіть важливіше – модульність набагато спрощує оновлення або модифікацію програми, тому що її частини не перемішані. Коли ви отримаєте достатні знання про функції, то зможете помістити кожен модуль у функцію, покращуючи тим самим модульність програми.

5.1.3. Ініціалізація масивів

Масиви часто використовуються для зберігання даних, які необхідні для подальшої роботи програми. Наприклад, 12-ти елементний масив може зберігати кількість днів у кожному місяці. У випадках подібного роду зручно ініціалізувати масив на початку програми:

```
int
main(void)
{
    // ANSI C та подальші стандарти
```

```
    int powers[8] = { 1, 2, 4, 6, 8, 16, 32, 64 };
}
```

Нескладно помітити, що масив ініціалізується з застосуванням списку значень, які розділяються комами. Список обмежується фігурними дужками. За бажанням між комами та значеннями можна розміщувати пробіли, що і зроблено в даному прикладі. Першому елементу (**powers[0]**) присвоюється значення **1**, другому (**powers[1]**) – значення **2** і т. д.

Розглянемо програму, яка виводить кількість днів у кожному місяці.

```
#include <stdio.h>
#include <windows.h>
#define MONTHS 12

int main(void)
{
   int days[MONTHS] = { 31, 28, 31, 30, 31, 30,
                        31, 31, 30, 31, 30, 31 };
   int index;

   SetConsoleOutputCP(1251);

  for(index = 0; index < MONTHS; index++)
  {
    switch(index + 1)
    {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12: printf("Місяць %02d має %2d день.\n",
                    index+1, days[index]);
             break;
    default: printf("Місяць %02d має %2d днів.\n",
             index+1, days[index]);
             break;
    }
  }
 return 0;
}
```

Результат виконання програми наведено на рис. 5.4.

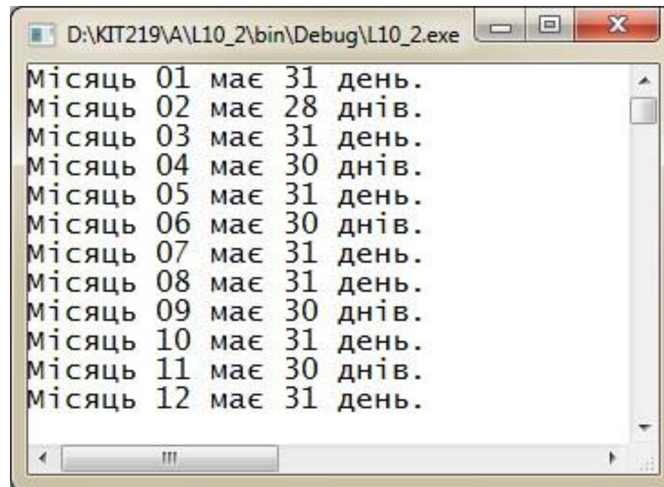


Рисунок 5.4 – Результат виконання програми, яка демонструє можливості ініціалізації масивів

Програма повідомляє некоректні відомості тільки для одного місяця один раз на чотири роки. Вона ініціалізує масив **days[]** списком розділених комами значень, які обмежуються фігурними дужками.

Зверніть увагу на те, що в цьому прикладі для представлення розміру масиву використовується символічна константа **MONTHS**. Це найпоширеніша і рекомендована практика.

Іноді доводиться застосовувати масив, який призначений лише для читання. Тобто програма буде діставати з нього значення, але не намагатися записувати нові значення в цей масив. В подібних випадках слід використовувати ключове слово **const** під час оголошення та ініціалізації масиву. Таким чином, в попередній програмі краще вказати наступне оголошення:

```
const int days[MONTHS] = { 31, 28, 31, 30, 31, 30,
                           31, 31, 30, 31, 30, 31 };
```

Це змушує програму трактувати кожен елемент масиву як константу. Для ініціалізації даних **const** треба застосовувати оголошення, оскільки через наявність **const** присвоїти їм значення пізніше не вийде.

В наступній програмі показано, що відбудеться, якщо не виконати ініціалізацію. Текст програми буде мати такий вигляд:

```
#include <stdio.h>
#define SIZE 4

int main(void)
{
     int no_data[SIZE]; // масив без ініціалізації
     int i;

      printf("===================\n");
      printf("%2s%14s\n", "i", "no_data[i]");
      printf("===================\n");
      for(i = 0; i < SIZE; i++)
          printf("%2d%14d\n", i, no_data[i]);
      printf("===================\n\n");
      return 0;
```

На рис. 5.5 наведено результат одного пробного запуску програми (ваші результати можуть не збігатися з даними на рисунку).

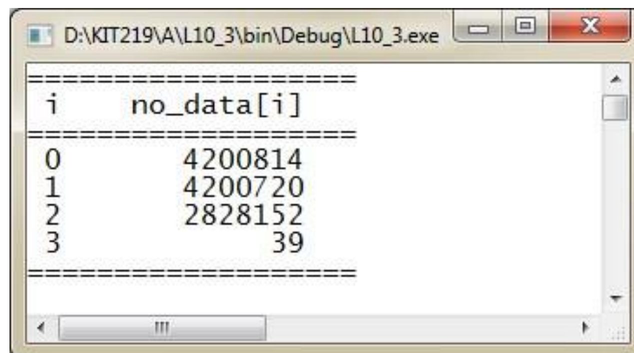


Рисунок 5.5 – Результат виконання програми, якщо ініціалізація масиву не проводилася

Елементи масиву схожі на звичайні змінні – якщо ви їх не ініціалізуєте, вони можуть мати будь-які значення. Компілятору дозволено просто брати значення, які вже находяться у відповідних комірках пам'яті. Саме за цієї причини ваші результати можуть не збігатися з отриманими на рис. 10.5.

Кількість елементів у списку повинна відповідати розміру масиву. Але що відбудеться, якщо розрахунок було проведено неправильно? Скоротимо в останній програмі список ініціалізації до двох елементів. Текст програми буде мати такий вигляд:

```
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int no_data[SIZE] = { 1428, 3754 };

    printf("===================\n");
    printf("%2s%14s\n", "i",
    "no_data[i]");
    printf("===================\n");
    for(int i = 0; i < SIZE; i++)
        printf("%2d%14d\n", i, no_data[i]);
    printf("===================\n\n");
    return 0;
}
```

Результат виконання програми наведено на рис. 5.6.

```
D:\KIT219\A\L10_4\bin\Debug\L10_4.exe
====================
 i     no_data[i]
====================
 0           1428
 1           3754
 2              0
 3              0
====================
```

Рисунок 5.6 – Результат виконання програми, якщо ініціалізація масиву проводилася частково

Як можна побачити з рис. 10.6, у компілятора не виникло жодних проблем. Коли значення в списку закінчилися, він ініціалізував інші елементи значенням 0. Іншими словами, якщо взагалі не ініціалізувати масив, його елементи, подібно звичайним змінним, отримають випадкові значення з пам’яті. Але якщо ініціалізувати масив частково, то елементи, що залишилися, встановлюються в 0. Можна дозволити компілятору привести розмір масиву у відповідність до списку, нічого не вказуючи в квадратних дужках:

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

int main(void)
{
   int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31 };
```

```
   int index;

   SetConsoleOutputCP(1251);

    for(index = 0; index < sizeof days / sizeof days[0]; index++)
    {
      switch(index + 1)
      {
         case 1:
         case 3:
         case 5:
         case 7:
         case 8:
         case 10: printf("Місяць %02d має %2d день.\n",
                            index+1, days[index]);
                  break;
          default: printf("Місяць %02d має %2d днів.\n",
                            index+1, days[index]);
                   break;
       }
     }
 return 0;
}
```

Необхідно зазначити два основних моменти.

Коли для ініціалізації масиву використовуються порожні квадратні дужки, компілятор підраховує кількість елементів у списку та встановлює розмір масиву автоматично.

Зверніть увагу на те, що зроблено в керуючому операторі циклу **for**. Через відсутність впевненості у можливості коректного підрахунку кількості елементів, програма самостійно визначає розмір масиву. Операція **sizeof** видає розмір в байтах наступного за нею об’єкта або типу. Таким чином, **sizeof days** – це розмір у байтах усього масиву, а **sizeof days[0]** – розмір в байтах одного елемента. Розділивши розмір усього масиву на розмір одного елемента, отримаємо кількість елементів у масиві.

Результат виконання програми наведено на рис. 5.7.

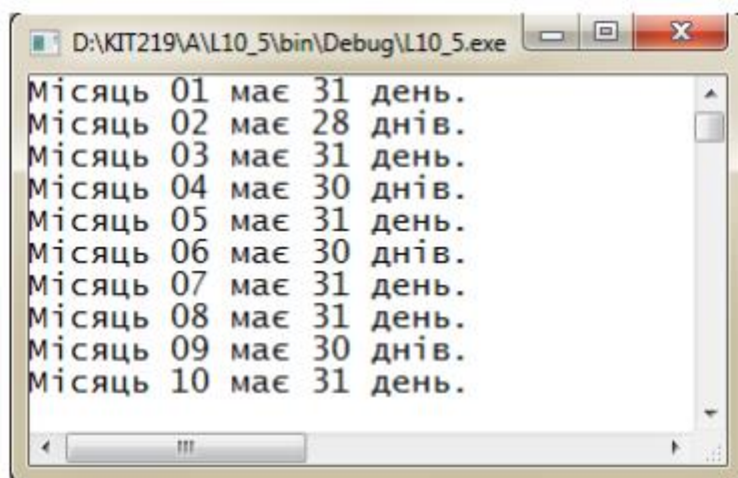


Рисунок 5.7 – Результат виконання програми, якщо кількість елементів масиву визначається автоматично

5.1.4. Призначені ініціалізатори (C99)

В стандарті **C99** додана нова можливість – **призначені ініціалізатори**. Цей засіб дозволяє обирати, які саме елементи будуть ініціалізовані. Припустимо, наприклад, що необхідно ініціалізувати тільки останній елемент в масиві. За допомогою традиційного синтаксису ініціалізації мови **C** знадобиться також ініціалізувати усі елементи, що передують останньому:

```
int mas[6]  = { 0, 0, 0, 0, 0, 212);   // традиційний синтаксис
```

Стандарт **C99** дозволяє застосовувати в списку ініціалізації індекс у квадратних дужках, щоб вказати конкретний елемент:

```
// ініціалізація елемента mas[5] значенням 212
int mas[6] = { [5] = 212 };
```

Як і у випадку звичайної ініціалізації, після того як буде проініціалізований хоча б один елемент, інші елементи, що залишилися не ініціалізованими, встановлюються в 0. Розглянемо більш складний приклад:

```
#include <stdio.h>
#define MONTHS 12

int main(void)
{
  int days[MONTHS] = { 31, 28, [4] = 31, 30, 31, [1] = 29 };
```

```
   int i;

   for(i = 0; i < MONTHS; i++)
      printf("%2d  days[%2d] = %2d\n", i + 1, i, days[i]);
   printf("\n");
   return 0;
}
```

Результат виконання програми для випадку, коли компілятор підтримує цей засіб **C99**, наведено на рис. 5.8.

```
D:\KIT219\A\L10_6\bin\Debug\L10_6.exe
 1  days[ 0] = 31
 2  days[ 1] = 29
 3  days[ 2] =  0
 4  days[ 3] =  0
 5  days[ 4] = 31
 6  days[ 5] = 30
 7  days[ 6] = 31
 8  days[ 7] =  0
 9  days[ 8] =  0
10  days[ 9] =  0
11  days[10] =  0
12  days[11] =  0
```

Рисунок 5.8 – Результат виконання програми, якщо компілятор підтримуєв списку ініціалізації індекс у квадратних дужках (C99)

Рисунок. 5.8 відображає декілька важливих характеристик призначених ініціалізаторів. По-перше, якщо за призначеним ініціалізатором знаходиться код з подальшими значеннями, як у послідовності **[4] = 31, 30, 31**, то ці значення використовуються для ініціалізації наступних елементів. Тобто, після ініціалізації **days[4]** значенням **31**, цей код ініціалізує **days[5]** і **days[6]** значеннями **30** і **31**, відповідно. По-друге, якщо код ініціалізує окремий елемент значенням більше одного разу, то актуальною буде остання ініціалізація.

Наприклад, у попередній програмі на початку списку проводиться ініціалізація **days[1]** значенням **28**, але пізніше це значення

перевизначено призначеним ініціалізатором **[1] = 29**.

А що відбудеться, якщо не вказати розмір масиву?

```
int stuff[] = { 1, [6] = 23 }; // що відбувається?
int staff[] = { 1, [6] = 4, 9, 10 }; // що відбувається?
```

Компілятор зробить масив достатньо великим, щоб вмістити значення ініціалізації. Так, масив **stuff** буде мати сім елементів з номерами **0-6**, а масив **staff** – на два елементи більше, тобто **9** елементів.

5.1.5. Присвоювання значень елементам масиву

Після того, як масив буде оголошений, елементам масиву можна присвоювати значення з використанням їх індексів. Наприклад, в наступній програмі елементам масиву присвоюються парні числа:

```
#include <stdio.h>
#define SIZE 10

int main(void)
{
   int counter, evens[SIZE];

   for(counter = 0; counter < SIZE; counter++)
   {
   evens[counter] = 2 * counter;
   printf("evens[%d] = %3d\n", counter, evens[counter]);
   }
   printf("\n");
   return 0;
}
```

Результат виконання програми наведено на рис. 10.9.

Зверніть увагу, що в коді використовується цикл для поелементного присвоювання значень. Мова Сне дозволяє присвоювати один масив іншому як єдиний модуль. Крім того, неможна застосовувати форму списку в фігурних дужках, окрім як при ініціалізації.

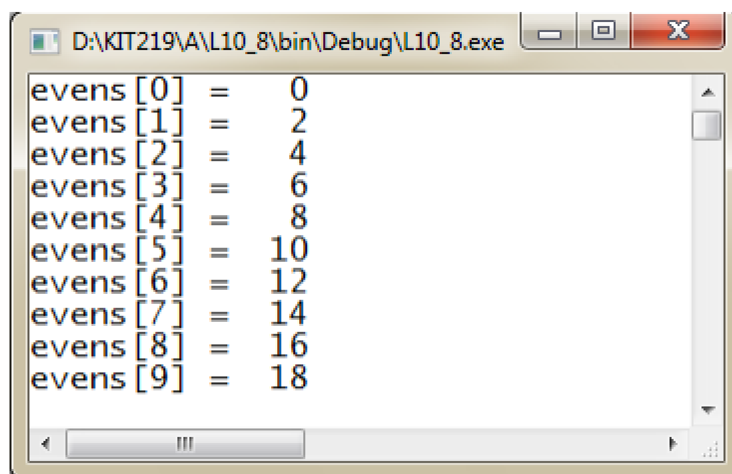


Рисунок 5.9 – Результат виконання програми для демонстрації можливості присвоювання значень елементам масиву

5.1.6. Межі масиву

Ви повинні забезпечити, щоб індекси масиву не виходили за межі, тобто переконатися в тому, що вони мають значення, які є допустимими для масиву. Наприклад, припустимо, що існує наступне оголошення:

```
int mas[20];
```

Після цього саме на вас покладається відповідальність за те, що в програмі будуть застосовуватися тільки індекси з діапазону від **0** до **19**. Тобто компілятор не зобов'язаний вас перевіряти. Хоча деякі компілятори і будуть попереджати вас про наявність проблем, але вони все одно будуть продовжувати компіляцію програми у будь-якому випадку.

Розглянемо програму, яка створює масив з чотирма елементами, а потім недбало використовує значення індексу в діапазоні від **-1** до **6**.

```
#include <stdio.h>
#include <windows.h>
#define SIZE 4

int main(void)
{
   int value1 = 44;
   int arr[SIZE];
   int value2 = 88;
   int i;
```

```
    SetConsoleOutputCP(1251);

    printf("value1 = %d, value2 = %d\n\n", value1, value2);
    for(i = -1; i <= SIZE; i++)
        arr[i] = 2 * i + 1;
    for(i = -1; i < 7; i++)
         printf("%2d %8d\n", i, arr[i]);
    printf("\nvalue1  =  %d,  value2  =  %d\n",  value1,  value2);
    printf("адреса arr[-1]: %p\n", &arr[-1]);
    printf("адреса arr[4]:  %p\n", &arr[4]);
    printf("адреса value1:  %p\n", &value1);
    printf("адреса value2:  %p\n", &value2);
    return 0;
}
```

Компілятор не перевіряє припустимість індексів. В стандарті мови **C** результат застосування некоректного індексу є невизначеним. Це означає, що після запуску програма може виглядати працездатною, функціонувати дивним чином або зовсім аварійно завершитися. Нижче наведено результат виконання програми при використання компілятора **GCC** (рис. 5.10).

```
D:\KIT219\A\L10_9\bin\Debug\L10_9.exe
value1 = 44, value2 = 88

-1       -1
 0        1
 1        3
 2        5
 3        7
 4        9
 5        5
 6  4200864

value1 = 9, value2 = -1
адреса arr[-1]: 0022FEF4
адреса arr[4]:  0022FF08
адреса value1:  0022FF08
адреса value2:  0022FEF4
```

Рисунок 5.10 – Результат виконання програми, якщо компілятор не контролює вихід за межі масиву

Зверніть увагу на те, що компілятор зберіг значення **value1** безпосередньо після масиву, а значення **value2** – прямо перед ним. Інші компілятори можуть зберігати дані в пам’яті в іншому порядку. В даному випадку, як показано на рис. 10.8, **arr[-1]** відповідає тій самій комірці

пам’яті, що й **value2**, а **arr[4]** – тій самій комірці пам’яті, що й **value1**. Тобто застосування індексів, які виходять за межі масиву, призводить до того, що програма модифікує значення інших змінних. Інший компілятор може дати інші результати, включаючи аварійне завершення програми.

Може виникнути питання, чому в **C** подібне дозволено. Це є наслідком прийнятої в мові філософії довіри до програміста. Відсутність перевірки меж дозволяє програмі на **C** виконуватися швидше. Компілятор не завжди здатний виявити всі помилки індексації, оскільки значення індексу може залишатися невизначеним до тих пір, поки не почнеться виконання програми. За цією причиною для забезпечення безпеки компілятору прийшлося б додавати додатковий код для перевірки кожного індексу під час виконання, що призводило б до зниження швидкості виконання. Таким чином, компілятор **C** довіряє програмісту через те, що він коректно кодує, і нагороджує його більш швидкою програмою. Звичайно, не всі програмісти заслуговують такої довіри, і в таких випадках можуть виникати проблеми.

Запам’ятайте одну простую річ: нумерація в масиві починається з 0. Необхідно напрацювати звичку використовувати символічну константу в оголошенні масиву та в інших місцях, де застосовується розмір масиву. Це допоможе забезпечити погоджене використання розміру масиву в програмі.

5.1.7. Зазначення розміру масиву

До виходу стандарту **C99** при оголошенні масиву в квадратних дужках треба було поміщати константний цілочисловий вираз, сформований з цілочислових констант. В цьому сенсі вираз **sizeof** вважається цілочисловою константою, але значення **const** – не існує. Крім того, значення такого виразу повинне було бути більше 0. Починаючи зі стандарту **C99**, в мові **C** допускаються оголошення наступного виду:

```
int n = 5;
int m = 8;
float a8[n];
float a9[m];
```

Вони призводять до створення нового виду масивів, які мають назву **масивів змінної довжини**. В стандарті **C11** відступили від цієї сміливої ініціативи, зробивши масиви змінної довжини додатковою, а не обов'язковою мовною можливістю.

Масиви змінної довжини були введені у стандарт **C99** головним чином, щоб надати можливість **C** стати кращою мовою в плані числових обчислювань. Наприклад, масиви змінної довжини полегшують перетворення існуючих бібліотек підпрограм цифрових розрахунків на мові **FORTRAN** в код на мові **C**. Масиви змінної довжини мають низку обмежень: наприклад, масив змінної довжини неможливо ініціалізувати під час його оголошення.

Розглянемо програму, яка демонструє приклад використання масиву змінної довжини. Текст програми має такий вигляд:

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

int main(void)
{
  int n;

  SetConsoleOutputCP(1251);

  printf("Введіть розмірність масиву: ");
  scanf("%d", &n);

  int mas[n];      // Масив змінної довжини оголошується
                   //лише після введення розмірності n

  srand(time(NULL));

  printf("\nЕлементи масиву\n");
  for(int i = 0; i < n; i++)
  {
        mas[i] = rand() % 10;
        printf("%2d", mas[i]);
  }
  printf("\n\n");
  return 0;
}
```

Результат виконання програми для розмірностей n = 7 і n = 14 масиву **`mas`** наведені на рис. 5.11 і рис. 5.12 відповідно.

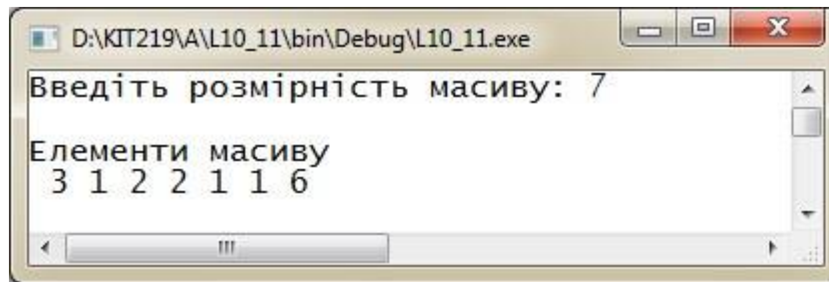


Рисунок 5.11 – Приклад використання масиву змінної довжини для n = 7

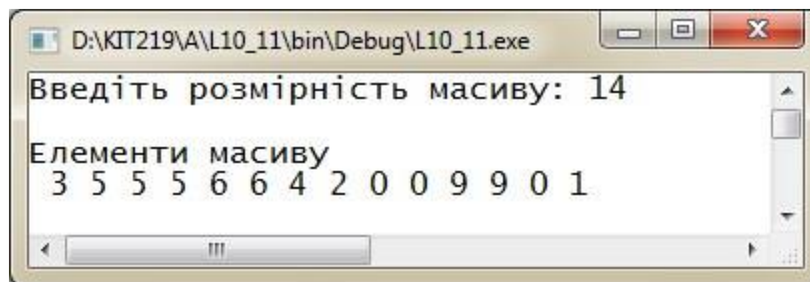


Рисунок 5.12 – Приклад використання масиву змінної довжини для n = 14

5.1.8. Двовимірні масиви

Якщо метеоролог бажає проаналізувати дані про опади за останні п’ять років, першим ділом йому необхідно обрати спосіб представлення даних. Один з варіантів передбачає використання **`60`** змінних, по одній для кожного елемента даних. Масив з **`60`** елементів здається більш досконалим способом, але набагато краще зберігати дані для кожного року окремо. Можна було б застосувати **`5`** масивів по **`12`** елементів кожний, але це грубий підхід, який перетвориться на важко вирішувану проблему.

Більш ефективний підхід передбачає використання масиву масивів. Головний масив повинен мати п’ять елементів, по одному на кожний рік. У свою чергу, кожен з цих елементів є 12-ти елементним масивом (по одному елементу на кожен місяць). Такий масив оголошується наступним чином:

```
// масив з 5 масивів по 12 елементів типу float
float rain[5][12];
```

Внутрішня частина rain[5] говорить про те, що **rain** це масив з п’ятьма елементами. Частина, що залишилася (**float [12]**), інформує про те, що кожен елемент має тип **float**, тобто кожен з п’яти елементів **rain** сам по собі є масивом з **12** значень типу **float**.

Відповідно до цієї логіки, **rain[0]**, будучи першим елементом масиву **rain**, являє собою масив з **12** значень типу **float**. Те ж саме стосується **rain[1]**, **rain[2]** і т. д. Якщо **rain[0]** являє собою масив, то його першим елементом буде **rain[0][0]**, другим елементом – **rain[0][1]** і т. д. Іншими словами, **rain** – це **5**-ти елементний масив з **12**-ти елементних масивів **float**, **rain[0]** – масив з **12** елементів типу **float**, а **rain[0][0]** – значення типу **float**. Для доступу, наприклад, до значення в рядку **2** і стовпці **3** застосовується запис **rain[2][3]**.

Масив **rain** можна представити у вигляді двовимірного масиву, що складається з п’яти рядків, кожен з яких містить **12** стовпців (рис. 5.13). Змінюючи другий індекс, можна переміститися по рядку, місяць за місяцем. Змінюючи перший індекс, можна перейти вертикально вздовж стовпця, рік за роком.

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]	[0][6]	[0][7]	[0][8]	[0][9]	[0][10]	[0][11]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]	[1][6]	[1][7]	[1][8]	[1][9]	[1][10]	[1][11]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]	[2][6]	[2][7]	[2][8]	[2][9]	[2][10]	[2][11]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]	[3][6]	[3][7]	[3][8]	[3][9]	[3][10]	[3][11]
[4][0]	[4][1]	[4][2]	[4][3]	[4][4]	[4][5]	[4][6]	[4][7]	[4][8]	[4][9]	[4][10]	[4][11]

Рисунок 5.13 – Представлення **rain** у вигляді двовимірного масиву, де в кожному елементі вказані лише індекси

Двовимірне представлення – це всього лише зручний спосіб візуалізації масиву з двома індексами. В пам’яті комп’ютера такий масив буде зберігатися послідовно, починаючи з першого 12-ти елементного масиву, за яким йде

другий 12-ти елементний масив, і т. д.

Скористаємося цим двовимірним масивом у програмі обробки погодних даних. Мета програми полягає у знаходженні загальної суми опадів для кожного року, середніх значень опадів за рік і середніх значень опадів за місяць. Щоб обчислити загальну суму опадів за рік, необхідно додати усі дані в окремому рядку. Щоб отримати середнє значення опадів за конкретний місяць, знадобиться скласти всі значення в заданому стовпці. Двовимірний масив спрощує візуальне представлення та виконання цих дій. Програма має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#define MONTHS 12          // кількість місяців у році

#define YEARS 5            // кількість років, для яких доступні дані

int
main(void)
{
   // ініціалізація даними про опади за період з 2014 по 2018 рр.
   const float rain[YEARS][MONTHS] =
    {
       { 4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6 },
       { 8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3 },
       { 9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4 },
       { 7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2 },
       { 7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2 }
    };
    int year, month;
    float subtot, total;

    SetConsoleOutputCP(1251);

    printf("  РІК   КІЛЬКІСТЬ ОПАДІВ (в дюймах)\n");
    for(year = 0, total = 0; year < YEARS; year++)
    {
       //для кожного року додавати кількість опадів за кожен місяць
       for(month = 0, subtot = 0; month < MONTHS; month++)
            subtot += rain[year][month];
       printf("%5d %15.1f\n", 2014 + year, subtot);
       total += subtot; // Загальна сума для всіх років
    }
    printf("\nСередньорічна кількість опадів складає %.1f "
           "дюймів.\n\n", total/YEARS);
    printf("СЕРЕДНЬОМІСЯЧНА КІЛЬКІСТЬ ОПАДІВ:\n\n");
    printf("   Січ   Лют   Бер  Квіт  Трав   Чер   Лип  Серп"
```

```
            "   Вер  Жовт  Лист  Груд \n");
    for(month = 0; month < MONTHS; month++) {
        // для кожного місяця підсумовувати кількість
        // опадів протягом 5 років
        for(year = 0, subtot = 0; year < YEARS; year++)
            subtot += rain[year][month];
        printf("%6.1f", subtot/YEARS); }
    printf("\n");
    return 0;
}
```

Результат роботи програми наведено на рис. 5.14.

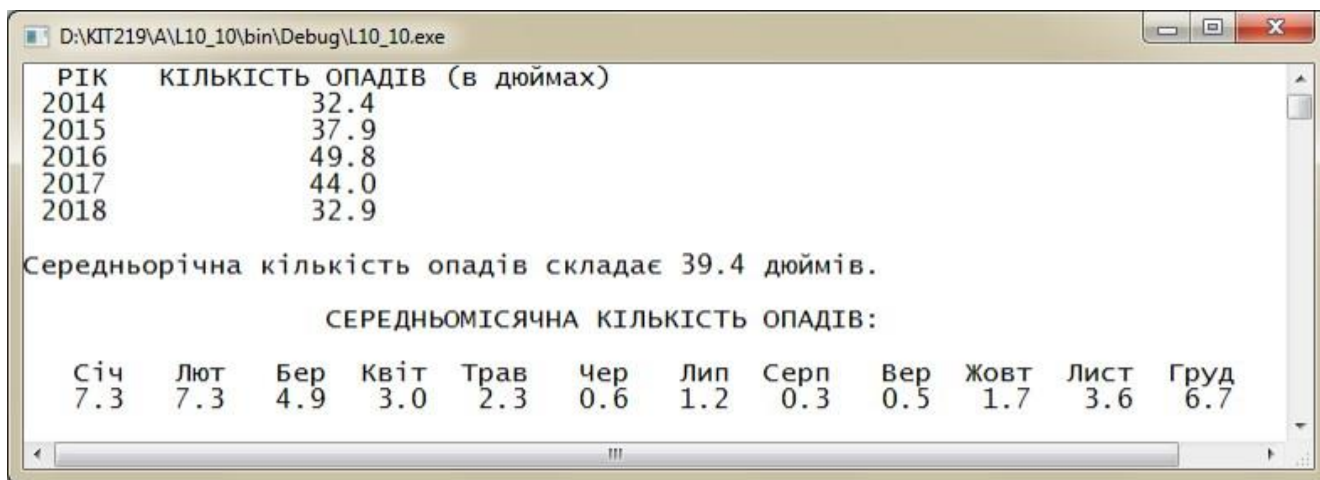


Рисунок 5.14 – Результат виконання програми для демонстрації роботи з двовимірними масивами

5.1.9. Ініціалізація двовимірного масиву

Ініціалізація двовимірного масиву побудована на прийомі, що застосовується для ініціалізації одновимірного масиву. Перш за все, згадайте, що ініціалізація одновимірного масиву виглядає наступним чином:

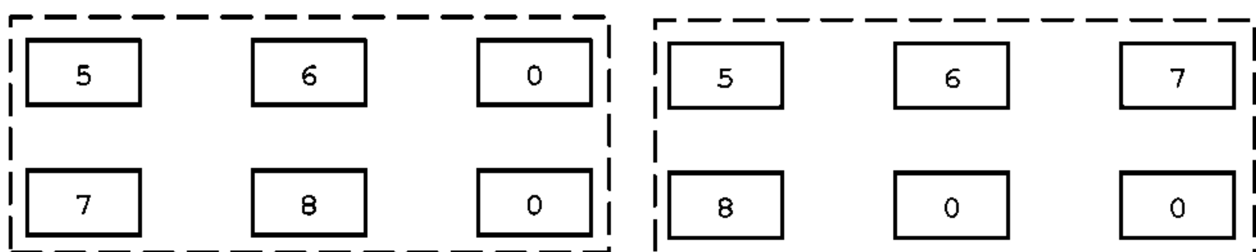
```
sometype arr[5] = { val1, val2, val3, val4, val5 };
```

В цьому оголошенні **val1**, **val2** і т. д. є значеннями типу **sometype**. Наприклад, якщо б типом **sometype** був тип **int**, то значенням **val1** могло бути число **7**, а якщо б типом **sometype** був тип **double**, то значенням **val1** могло бути число **11.34**. Але ж **rain** – це 5-ти елементний масив, кожен елемент якого є масивом, що складається з **12** значень типу **float**.

В ініціалізації масиву **rain** використовується п'ять списків чисел, які обмежені фігурними дужками і поміщені у зовнішній набір фігурних дужок. Дані з першої внутрішньої пари фігурних дужок присвоюються першому рядку масиву, дані з другої внутрішньої пари дужок – другому рядку масиву і т. д. Розглянуті вище правила відносно невідповідностей між кількістю даних і розміром масиву, застосовуються до кожного рядка. Це означає, що якщо внутрішній набір фігурних дужок містить **10** чисел, то він впливає лише на початкові **10** елементів у першому рядку. Останні два елемента в цьому рядку за замовчуванням ініціалізуються нулем. Якщо чисел задано надто багато, виникає помилка. Числа не переносяться на наступний рядок.

Внутрішні фігурні дужки можна було б не вказувати, залишивши тільки дві зовнішні дужки. При правильній кількості записів результат буде таким самим. Однак якщо записів недостатньо, масив заповнюється послідовно, рядок за рядком, поки дані не закінчаться. Потім елементи, що залишилися, будуть ініціалізуватися значенням **0**. На рис. 5.15 продемонстровані обидва способи ініціалізації масиву.

Оскільки масив **rain** містить дані, які не повинні бути модифіковані, під час оголошення масиву в програмі використовується модифікатор **const**




```
int sq[2][3] = { {5,6},{7,8} };
int sq[2][3] = { 5,6,7,8 };
```

Рисунок 5.15 – Два методи ініціалізації масиву

5.1.10. Масиви з великою кількістю вимірювань

Все, що було сказано про двовимірні масиви, можна поширити на тривимірні масиви та на масиви з великою кількістю вимірів. Тривимірний

масив оголошується наступним чином:

```
int box[10][20][30];
```

Одновимірний масив можна представити як рядок даних, двовимірний масив – як таблицю даних, а тривимірний масив – як декілька таблиць даних.

Наприклад, про масив **box** можна говорити як про **10** двовимірних масивів (кожен розміром **20**×**30**), які розташовані один на одному.

Масив **box** можна по-іншому представляти як масив масивів, що складається з масивів. Тобто це **10**-ти елементний масив, елементами якого є **20**-ти елементні масиви. Кожен **20**-ти елементний масив містить елементи, які являють собою **30**-ти елементні масиви. Або ж масиви можна просто розглядати з точки зору кількості необхідних індексів.

Зазвичай для обробки тривимірного масиву застосовуються три вкладені цикли, для обробки чотиривимірного масиву – чотири вкладені цикли і т. д.

Контрольні запитання та завдання

1. В якмх випадках в мові *C* використовують об’єкти, які звуться масивами?

2. Дайте визначення наступних понять: масив, елемент масиву, адреса масиву, розмір масиву та розмір елента масиву.

3. З якого числа починається нумерація елементів масиву?

4. До якого типу даних може стосуватися масив?

5. Як звуться числа, які застосовуються для ідентифікації елементів масиву?

6. В чому полягає особливість використання циклів **for** з масивами?

7. Як відбувається Ініціалізація масивів?

8. Для чого застововуються призначені ініціалізатори згідно стандарта **C99**?

9. Як відбувається присвоювання значень елементам масиву?

10. Що ви розумієте під межами масиву?

11. Наіщо в у стандарті `C99` мові С були введені масиви змінної довжини?

12. Дайте визначення поняття двовимірного масиву та як відбувається його Ініціалізація.

13. Як оголошується тривимірний масив?

14. Скільки необхідно застосувати вкладених циклів для обробки чотиривимірного масиву?

Завдання для самостійного розв'язання

1. Відсортуйте за неубуванням ($mas[i] \le mas[i+1]$) одновимірний цілочисловий масив методом «бульбашки», заданий випадковими цілими числами на проміжку [–99, 99]. Виведіть на екран вхідний і відсортований масиви, а також вміст масиву на кожному кроці сортування.

2. Відсортуйте за незростанням ($mas[i] \ge mas[i+1]$) одновимірний цілочисловий масив методом простого вибору, заданий випадковими цілими числами на проміжку [0, 50]. Виведіть на екран на зразок вхідний і відсортований масиви.

3. Масив розміром $2m + 1$, де m – натуральне число, заповнений випадковими цілими числами в діапазоні [10, 50]. Знайдіть у масиві медіану.

4. Напишіть С-програму, в якій при оголошенні символьного масиву відбувається його ініціалізація рядками символів.

5. Напишіть С-програму, яка заповнює двовимірний масив розмірністю 7×7 двозначними довільними цілими числами, а потім присвоює за допомогою циклів нулі довільним коміркам (на ваш розсуд). На консоль виведіть поруч вхідний та вихідний масиви.

6. ФУНКЦІЇ ТА ВКАЗІВНИКИ

6.1. Функції

Проектна філософія **C** передбачає використання функцій в якості основних будівельних блоків при створенні програм. Ви вже використовували стандартну бібліотеку **C**, коли застосовували такі функції, як **printf()**, **scanf()**, **getchar()**, **putchar()**, математичні функції та інші, не кажучи вже про головну функцію **main()**. Настав час перейти до створення своїх власних функцій. Слід зазначити, що деякі аспекти цього процесу вже були задіяні раніше. Проте тепер уся інформація, яка була отримана до того, буде об’єднана та розширена.

6.1.1. Поняття функції

Що ж собою являє функція? Функція – це самодостатня одиниця коду програми, яка спроектована для виконання окремої задачі. Структура функції та способи її можливого використання визначаються синтаксичними правилами. В мові **C** функція відіграє ту ж саму роль, яку в інших мовах програмування відіграють функції, підпрограми та процедури, хоча деталі можуть відрізнятися. Деякі функції призводять до виконання дії. Наприклад, функція **printf()** виводить дані на екран. Інші функції повертають значення, яке буде застосовуватися в програмі пізніше. Наприклад, функція **strlen()** повідомляє програму про довжину рядка. В загальному випадку функція може одночасно виконувати дії та повертати значення.

Чому ж необхідно використовувати функції? Перш за все, вони позбавляють від необхідності у багатократному написанні одного й того ж самого коду. Якщо в програмі треба виконувати певну задачу декілька разів, достатньо одного разу написати відповідну функцію. Потім цю функцію можна застосовувати в програмі там, де вона потрібна, або ж використовувати її в різних програмах, подібно тому, як в багатьох програмах була задіяна функція **putchar()**. Крім того, навіть якщо задача вирішується лише один раз в

одній окремій програмі, використання функції має сенс, оскільки це робить програму модульною, покращує її читабельність і спрощує внесення змін.

Припустимо, що треба написати програму, яка виконує наступні ії:

- читає список чисел;

- сортує ці числа;

- знаходить середнє значення цих чисел;

- виводить гістограму на екран.

В такому випадку можна було б написати наступну програму:

```
#include <stdio.h>
#define SIZE 50

int main(void)
{
    float list[SIZE];

    readlist(list, SIZE);
    sort(list, SIZE);
    average(list, SIZE);
    bargraph(list, SIZE);
    return 0;
}
```

Звичайно, вам доведеться також написати чотири функції **readlist()**, **sort()**, **average()** і **bargraph()**, але це вже деталі. Зазначені імена функцій пояснюють призначення та організацію програми. Потім з кожною функцією можна працювати окремо, поки вони не почнуть з успіхом вирішувати свої задачі. Після того, як ці функції будуть узагальнені, їх можна багатократно застосовувати в інших програмах.

Багато хто з програмістів надає перевагу думці про функцію як про «чорний ящик», що визначений в термінах інформації, яка до нього надходить (його ввід), і значення або дії, яку він виконує (його вивід). Вас не повинно турбувати те, що відбувається всередині чорного ящика, якщо тільки ви самі не займаєтесь розробкою цієї функції. Наприклад, коли ви використовуєте функцію **printf()**, то вам відомо, що необхідно передати

їй керувальний рядок і можливо деякі аргументи. Вам також відомо, що функція **printf()** повинна вивести. Однак ви зовсім не задумуєтесь про код, який реалізує **printf()**. Такий підхід по відношенню до функцій дозволяє зосередитися на загальній структурі програми, не відволікаючись на деталі. До того як розпочати написання коду, ретельно обдумайте, що повинна робити функція та в чому полягає її роль у програмі.

Що необхідно знати про функції? **Необхідно знати, як їх правильно визначати, викликати та забезпечувати взаємодію між ними.**

Почнемо з розгляду дуже простого прикладу, а потім будемо додавати до нього нові можливості, поки не буде отримана повна картина.

6.1.2. Створення та використання простої функції

Нашою першою метою буде створення функції, яка виводить в рядку **50** символів **'='**. Щоб надати цій функції сенсу, додамо її до програми, яка виводить простий заголовок для практичної роботи №7. Текст програми, що складається з функцій **main()** і **starbar()** має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#define WORK   "Практична робота №7"
#define NAME   "Отримання випадкових чисел"
#define AUTHOR "Виконав: студент групи КІТ-220а ЧЕРАБАЙ Ю.Ю."
#define WIDTH  50

void starbar(void);                    //  прототип функції starbar()
int main(void)
{
    SetConsoleOutputCP(1251);
starbar();                             // виклик функції starbar()
    printf("%s\n", WORK);
    printf("%s\n", NAME);
    printf("%s\n", AUTHOR);
starbar();                             // виклик функції starbar()

    return 0;
}
     void starbar(void)                // визначення функції starbar()
{
   int count;
   for(count = 1; count <= WIDTH; count++)
         putchar('=');
   putchar('\n');
```

}

Результат виконання програми наведено на рис. 6.1.

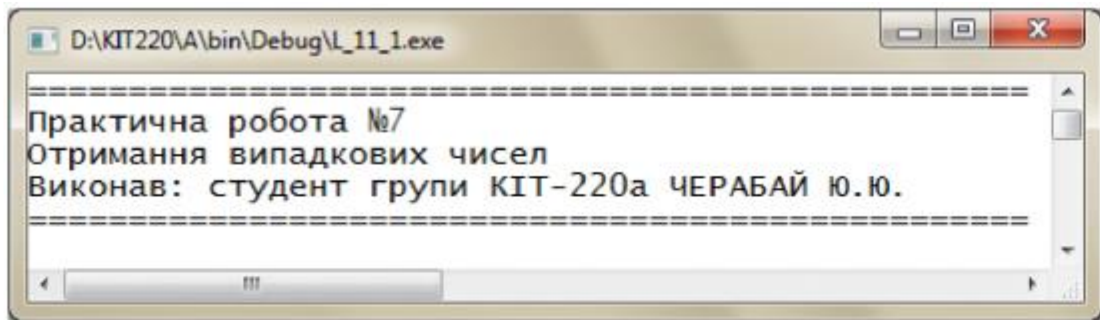


Рисунок 6.1 – Приклад застосування простої функції користувача

Розглянемо важливі аспекти цієї програми.

1) Ідентифікатор **starbar** застосовується в трьох окремих контекстах: **у прототипі функції**, який повідомляє компілятору різновид функції **starbar()**, **у виклику функції**, який призводить до виконання функції, і **у визначенні функції**, де в точності вказано все, що робить функція.

2) Подібно змінним, функції мають типи. В будь-якій програмі, в якій використовується функція, попередньо повинен бути оголошений тип цієї функції. Тому даний прототип **ANSI C** передує оголошенню функції **main()**:

```
void starbar(void);
```

Круглі дужки вказують, що **starbar** є іменем функції. Перше ключове слово **void** – це тип функції. Тип **void** говорить про те, що функція не повертає значення. Друге слово **void** (в круглих дужках) означає, що функція не приймає аргументів. Крапка з комою вказує на те, що функція оголошується, а не визначається. Тобто цей рядок повідомляє про те, що в програмі застосовується функція **starbar()**, яка не повертає значення і не приймає аргументів. Це означає, що компілятор повинен шукати її визначення десь в іншому місці.

3) У загальному випадку прототип вказує як тип значення, що

повертається функцією, так і типи аргументів, які вона очікує. Узагальнено цю інформацію називають **сигнатурою функції**. В даному конкретному випадку сигнатура вказує на те, що функція не має ні значення, яке повертається, ні аргументів.

4) Прототип функції **`starbar()`** розміщується в програмі попереду функції **`main()`**. Замість цього його можна було б помістити всередині функції **`main()`** там, де знаходяться оголошення змінних. Допустимо застосовувати будь-який з цих способів.

5) Головна програма викликає функцію (звертається до функції) **`starbar()`** з функції **`main()`**, для чого вказується її ім'я, круглі дужки та крапка з комою, створюючи таким чином оператор **`starbar();`**

Це форма виклику функції типу **`void`**. Кожного разу, коли керування програми зустрічає оператор **`starbar();`**, воно шукає функцію **`starbar()`** і виконує інструкції, які в ній знаходяться. По завершенні виконання коду **`starbar()`**, керування повертається на наступний рядок до функції, яка її викликала, – в даному випадку **`main()`**.

6) При визначенні функції **`starbar()`** у програмі застосовується та ж сама форма, що й при визначенні **`main()`**. Визначення починається з типу, імені та круглих дужок. Далі йде відкривальна фігурна дужка, оголошення змінних, оператори функції та закривальна фігурна дужка. Зверніть увагу на те, що після цього екземпляра функції **`starbar()`**, крапка з комою не вказується. Відсутність крапки з комою говорить компілятору про те, що функція **`starbar()`** визначається, а не викликається або вказується її прототип.

7) Функції **`starbar()`** і **`main()`** у програмі знаходяться в одному файлі. Їх можна також рознести по двох окремих файлах. Форму з одним файлом легше компілювати. Два окремих файли спрощують застосування однієї функції в різних програмах. Якщо ви розміщуєте функцію в окремому файлі, то повинні помістити до нього також необхідні директиви **`#define`** і

#include. Використання двох і більшої кількості файлів буде обговорюватися в одній з подальших лекцій, а поки всі функції будемо зберігати в одному файлі. Закривальна фігурна дужка **main()** вказує компілятору, де ця функція закінчується, а наступний за нею заголовок **starbar()** повідомляє компілятору про те, що **starbar()** є функцією.

8) Змінна **count** у функції **starbar()** є локальною. Це означає, що вона відома тільки цій функції. Ім'я **count** можна застосовувати в інших функціях, включаючи **main()**, і це не призведе до конфлікту. Просто в програмі будуть існувати окремі незалежні одна від одної змінні, які мають однакові імена.

Якщо думати про функцію **starbar()** як про чорний ящик, то її дія полягає у виводі рядка символів **'='**. Вона не приймає вхідні дані, оскільки їй не потрібна будь-яка інформація від функції, яка її викликає. Вона не надає (тобто не повертає) інформацію функції **main()**, тому **starbar()** не має значення, що повертається. Іншими словами, функція **starbar()** не має потреби в будь-якому обміні даними з функцією, яка її викликала.

Створимо тепер функцію, для якої такий обмін даними є необхідним.

6.1.3. Аргументи функції

Заголовок, який виводився на екран у попередній програмі, можна центрувати. Для цього слід розмістити відповідну кількість пробілів перед виводом рядка тексту. Така поведінка є аналогічною до функції **starbar()**, яка виводила задане число символів **'='**, але тепер необхідно виводити ще певну кількість пробілів. Замість написання окремої функції для кожної задачі створимо одну, але більш універсальну функцію, яка вирішує обидві задачі. Назвемо цю нову функцію **show_n_char()** (ім'я означає, що символ відображається **n** разів). Єдина зміна полягає в тому, що замість використання вбудованих значень для символу, що відображається, і кількості повторень, у функції **show_n_char()** для цього будуть застосовуватися аргументи.

Припустимо, що доступний простір має ширину **50** символів. Рядок з символів **'='** містить **50** символів, в точності відповідаючи за шириною, і виклик **show_n_char('=', 50)** повинен виводити цей рядок точно таким чином, як це раніше робила функція **starbar()**. Що можна сказати про пробіли, які використовуються для центрування рядка "Практична робота №7"? Рядок "Практична робота №7" має ширину **19** символів, тому в першій версії програми за заголовком йшов **31** пробіл. Для центрування рядка необхідно почати рядок з **15** пробілів, що надасть в результаті **15** пробілів з одного боку і **16** пробілів з іншого. Таким чином, можна застосовувати виклик **show_n_char(' ', 15)**.

Функція **show_n_char()** не додає символ нового рядка, як це робить **starbar()**,оскільки в цьому рядку може знадобиться вивести інший текст.

Модифікована версія програми має такий вигляд:

```
#include <stdio.h>
#include <string.h>                    //  для функції strlen()
#include <windows.h>
#define WORK   "Практична робота №7"
#define NAME   "Отримання випадкових чисел"
#define AUTHOR "Виконав: студент групи КІТ-220а ЧЕРАБАЙ Ю.Ю."
#define WIDTH 50
#define SPACE ' '

void show_n_char(char ch, int num);  //  прототип функції
int main(void)
{
    int spaces;
    SetConsoleOutputCP(1251);

    // використання констант в якості аргументів
    show_n_char('=', WIDTH);
    putchar('\n');
    // використання констант в якості аргументів
    show_n_char(SPACE, 15);
    printf("%s\n", WORK);
    // дозволити програмі обчислити кількість пробілів
    spaces = (WIDTH - strlen(NAME)) / 2;
    // використання констант в якості аргументів
    show_n_char(SPACE, spaces);
    printf("%s\n", NAME);
```

```
    show_n_char(SPACE, (WIDTH - strlen(AUTHOR)) / 2);
    printf("%s\n", AUTHOR);
    show_n_char('=', WIDTH); putchar('\n');
    return 0;
}
// визначення функції show_n_char()
void show_n_char(char ch, int num)
{
     for(int count = 1; count <= num; count++)
         putchar(ch);
}
```

Результат виконання модифікованої програми наведено на рис. 6.2.

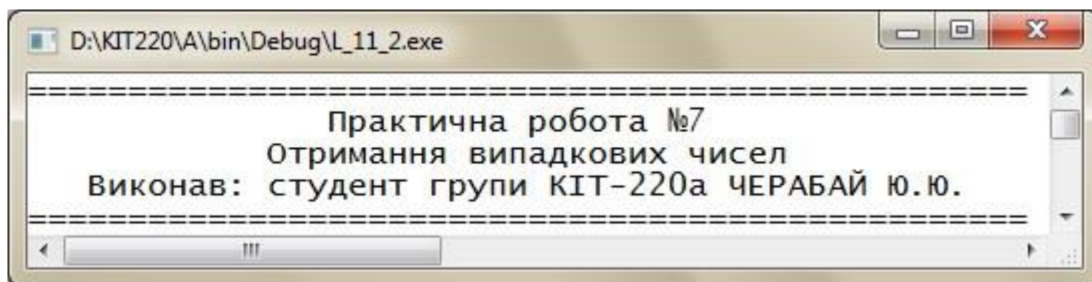


Рисунок 6.2 – Приклад використання простої функції з аргументами

6.1.4. Визначення функції з аргументами: формальні параметри

Визначення функції починається з наступного заголовка **ANSI C**:

```
void show_n_char(char ch, int num)
```

Цей рядок інформує компілятор про те, що функція **show_n_char()** приймає два аргументи з іменами **ch** і **num**. **ch** має тип **char**, а **num** – тип **int**. Змінні **ch** і **num** звуться формальними аргументами або формальними параметрами. Подібно змінним, що визначені всередині функції, формальні параметри є локальними змінними, які є закритими для функції. Це означає, що можна не турбуватися, якщо їх імена будуть дублюватися в інших функціях. Значення цим змінним будуть присвоюватися під час виклику функції. Зверніть увагу, що форма **ANSI C** вимагає, щоб кожній змінній передував її тип. Тобто, на відміну від звичайних оголошень, неможна застосовувати список змінних, які мають один і той самий тип:

```
void dibs(int x, y, z)          // некоректний заголовок функції
void dubs(int x, int y, int z) // правильний заголовок функції
```

Незважаючи на те, що функція `show_n_char()` приймає значення з `main()`, вона нічого не повертає, тому `show_n_char()` має тип `void`.

Тепер поглянемо, як користуватися цією функцією.

6.1.5. Створення прототипу функції з аргументами

Ми застосовуємо прототип **ANSI C**, щоб оголосити функцію перед її застосуванням:

```
void show_n_char(char ch, int num);
```

Коли функція приймає аргументи, прототип визначає їх кількість і типи, використовуючи розділений комами список типів. За бажанням імена змінних в прототипі можна не вказувати:

```
void show_n_char(char, int);
```

Застосування імен змінних в прототипі не призводить до створення цих змінних. Це просто прояснює той факт, що `char` означає змінну типу `char` та ін.

6.1.6. Виклик функції з аргументами: фактичні аргументи

Значення `ch` і `num` присвоюється з використанням фактичних аргументів у виклику функції. Розглянемо перший випадок застосування `show_n_char()`:

```
show_n_char(SPACE, 15);
```

Фактичними аргументами є `SPACE` і `15`. Ці значення присвоюються відповідним формальним параметрам функції `show_n_char()` – змінним `ch` і `num`. **Формальний параметр** – це змінна у функції, що викликається, а **фактичний аргумент** – це конкретне значення, яке функція, що викликає, присвоює змінній всередині функції, що викликається. Як показує приклад, фактичним аргументом може бути константа, змінна або навіть більш складний вираз. Незалежно від того, чим є

фактичний аргумент, він обчислюється, і його значення копіюється у відповідний формальний параметр для функції. Наприклад, розглянемо фінальне використання **show_n_char()**:

```
show_n_char(SPACE, (WIDTH - strlen(AUTHOR)) / 2);
```

Обчислення довгого виразу, який утворює другий фактичний аргумент, дає в результаті **2**. Потім значення **2** присвоюється змінній **num**. Функція не знає, та й не турбується про те, звідти потрапляє це число – з константи, змінної або більш загального виразу. Оскільки функція, що була викликана, працює з даними, скопійованими з функції, яка викликає, початкові дані у функції, що викликає, захищені від будь-яких маніпуляцій, які функція, що викликається, застосовує до їх копій.

6.1.7. Представлення функції у вигляді чорного ящика

При представленні функції **show_n_char()** у вигляді чорного ящика вхідними даними є символ, який необхідно вивести на екран, і кількість пробілів, які слід пропустити. Результатом буде вивід символу вказане число разів. Вхідні дані передаються функції за допомогою аргументів. Цієї інформації цілком достатньо для розуміння того, як ця функція використовується в **main()**. Крім того, ця інформація є проектною специфікацією для написання функції.

Той факт, що **ch**, **num** і **count** – це локальні змінні, які є закритими в межах функції **show_n_char()**, є суттєвим аспектом підходу з чорним ящиком. Якщо б у функції **main()** застосовувалися змінні з такими ж самими іменами, то це були б інші, незалежні змінні. Тобто, якщо б у **main()** була б змінна **count**, то зміна її значення не призвело б до зміни значення **count** у **shownchar()**, і навпаки. Все, що відбувається всередині чорного ящика, приховано від функції, що викликає.

6.1.8. Повернення значення з функції за допомогою return

Ви вже бачили, як передавати інформацію з функції, яка викликає, до функції, яка викликається. Для відправлення інформації у зворотному напрямку використовується значення, що повертається. Щоб нагадати, як це працює, реалізуємо функцію, яка повертає менше значення з двох аргументів. Назвемо цю функцію **imin()**, оскільки вона призначена для підтримки значень типу **int**. Крім того, створимо просту функцію **main()**, єдиною метою якої буде перевірка працездатності **imin()**. Програму, яка розроблена для такого тестування функцій, іноді звуть **драйвером**. Драйвер отримує функцію для перевірки. Якщо функція проходить перевірку успішно, її можна застосовувати в більш суттєвій програмі.

Код програми для драйвера та функції вибору мінімального значення має такий вигляд:

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

int imin(int, int);

int main(void) {
    int evil1, evil2;

    SetConsoleOutputCP(1251);

    printf("Введіть два цілих числа або q для завершення:\n");
    while(scanf("%d %d", &evil1, &evil2) == 2)
    {
        printf("Меншим з двох чисел %d і %d є %d.\n\n",
               evil1, evil2, imin(evil1, evil2));
        printf("Введіть два цілих числа або"
               " q для завершення:\n");
    }
    printf("Програма завершена.\n");
    return 0;
}

int imin(int n, int m)
{
    return (n < m) ? n : m;
}
```

Результат виконання програми наведено на рис. 6.3.

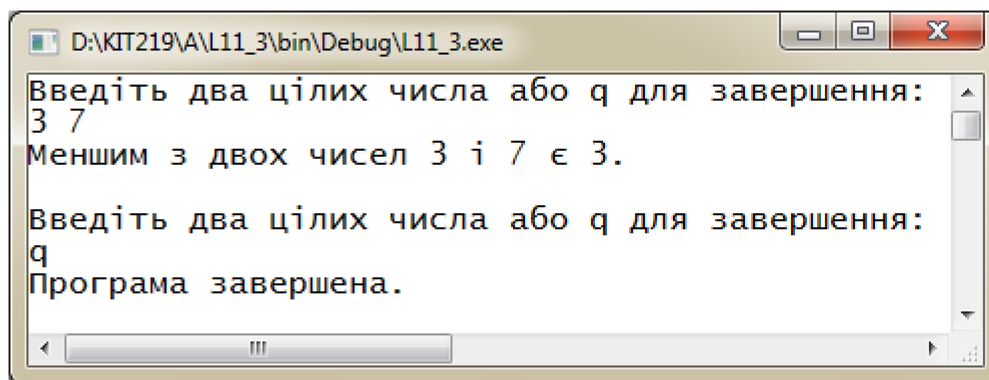


Рисунок 6.3 – Приклад використання функції визначення мінімального значення

Результатом обчислення умовного виразу в тілі функції `imin()` буде менше зі значень `n` і `m`, після чого дане значення повертається до функції, яка викликає. Якщо ви надаєте перевагу розміщувати значення, що повертається, у круглих дужках, можете так робити, хоча круглі дужки не є обов'язковими.

Можна також застосовувати оператор такого виду:

```
return;
```

Він призводить до завершення функції та повертає керування до функції, що викликає. Через те, що вираз за `return` відсутній, значення не повертається, і ця форма повинна використовуватися тільки у функціях типу `void`.

6.1.9. Типи функцій

Функції повинні оголошуватися із зазначенням типів. Функція зі значенням, що повертається, повинна бути оголошена з тим самим типом, що є у значення, яке повертається. Функція без значення, що повертається, повинна бути оголошена з типом `void`. Якщо функції не призначений тип, то в більш ранніх версіях `C` передбачалося, що така функція належить до типу `int`. Однак підтримка такого неявного передбачення про тип `int` зі стандарту `C99` виключена.

Оголошення типу є частиною визначення функції. Майте на увазі, що

воно відноситься до значення, що повертається, але не до аргументів функції. Наприклад, наступний заголовок функції вказує на те, що визначається функція, яка приймає два аргументи типу **int**, але повертає значення типу **double**:

```
double klink(int a, int b)
```

Для коректного використання функції програма повинна знати тип функції до її першого виклику. Один зі способів досягнення цього передбачає розміщення повного визначення функції до її першого застосування. Однак такий метод може ускладнити сприйняття програми. Крім того, функції можуть бути частиною бібліотеки **C** або знаходиться в будь-якому іншому файлі. Таким чином, зазвичай ви інформуєте компілятор про функції, оголошуючи їх заздалегідь.

В стандартній бібліотеці **ANSI C** функції згруповані в сімейства, кожне з яких має власний файл заголовку. Такі файли заголовку містять серед інших оголошення функцій в сімействах. Наприклад, файл заголовку **stdio.h** містить оголошення для стандартних бібліотечних функцій вводу-виводу, таких як **printf()** і **scanf()**. Файл заголовку **math.h** містить оголошення для множини математичних функцій, наприклад, оголошення

```
double sqrt(double);
```

яке повідомляє компілятору про те, що функція **sqrt()** приймає параметр типу **double** та повертає значення типу **double**. Не треба плутати ці оголошення з визначеннями. Оголошення функції інформує компілятор про те, який тип має функція, а визначення функції надає дійсний код. Включення файлу заголовка **math.h** повідомляє компілятор про те, що типом, який повертається, для **sqrt()** є **double**, але код для функції **sqrt()** знаходиться в окремому файлі бібліотечних функцій.

6.1.10. Створення прототипів функцій в ANSIC

Традиційна схема оголошення функцій, яка використовувалася до появи **ANSI C**, мала певний недолік, оскільки передбачала вказівку типу для значення функції, що повертається, але не для її аргументів. Давайте поглянемо, якого виду проблеми виникають у випадку оголошення функцій за старою формою.

Наступне оголошення, яке застосовувалося до виходу **ANSI C**, інформує компілятор про те, що функція **imin()** повертає значення типу **int**:

```
int imin();
```

Однак воно нічого не говорить про кількість або типи аргументів цієї функції. В результаті, якщо функція **imin()** використовується з некоректною кількістю аргументів або невідповідними їм типами, то компілятор не зможе виявити помилку.

6.1.11. Рішення стандарту ANSI C

Підхід до рішення проблем з невідповідністю аргументів, який реалізований в стандарті **ANSI C**, передбачає дозвіл зазначати в оголошенні функції також і типи змінних. Результатом є **прототип функції** – оголошення, в якому встановлюється тип, що повертається, кількість аргументів, а також їх типи. Щоб зазначити, що функція **imax()** потребує два аргументи **int**, її можна оголосити за допомогою одного з наступних прототипів:

```
int imax(int, int);
int imax(int a, int b);
```

В першій формі застосовується список типів, що розділені комами. В другій формі до типів додані імена змінних. Пам’ятаєте, що імена змінних є фіктивними та не зобов’язані відповідати іменам, що використовуються у визначенні функції.

Маючи цю інформацію, компілятор може перевірити, чи збігається

виклик функції з прототипом. Чи правильно вказана кількість аргументів? Чи мають вони коректні типи? У випадку, коли типи не збігаються, коли обидва типи є числовими, компілятор перетворює значення фактичних аргументів до типів формальних параметрів. Наприклад, виклик `imax(3.0, 5.0)` стає викликом `imax(3, 5)`.

Розглянемо програму з застосуванням прототипу функції `imax()`. Текст програми має такий вигляд:

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

int imax(int, int);       // прототип

int main(void)
{
    SetConsoleOutputCP(1251);
    printf("Найбільшим значенням з %d i %d є %d.\n",
           3, 5, imax(3));
    printf("Найбільшим значенням з %d i %d є %d.\n",
           3, 5, imax(3.0, 5.0));
    return 0;
}

int imax(int n, int m)    // визначення функції
{
    return (n > m ? n : m);
}
```

Спроба компіляції програми призведе до видачі компілятором повідомлення про помилку, яка вказує на те, що виклик `imax()` містить надто мало параметрів (рис. 6.4).

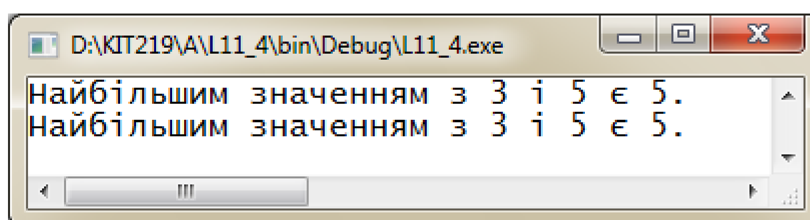


Рисунок 6.4 – Повідомлення компілятора про помилку у виклику функції

А що можна сказати про помилки, які пов'язані з типами? Для їх дослідження замінимо `imax(3)` викликом `imax(3, 5)` і спробуємо скомпілювати програму ще раз. Повідомлень про помилки не буде. Результат виконання програми наведено на рис. 6.5.

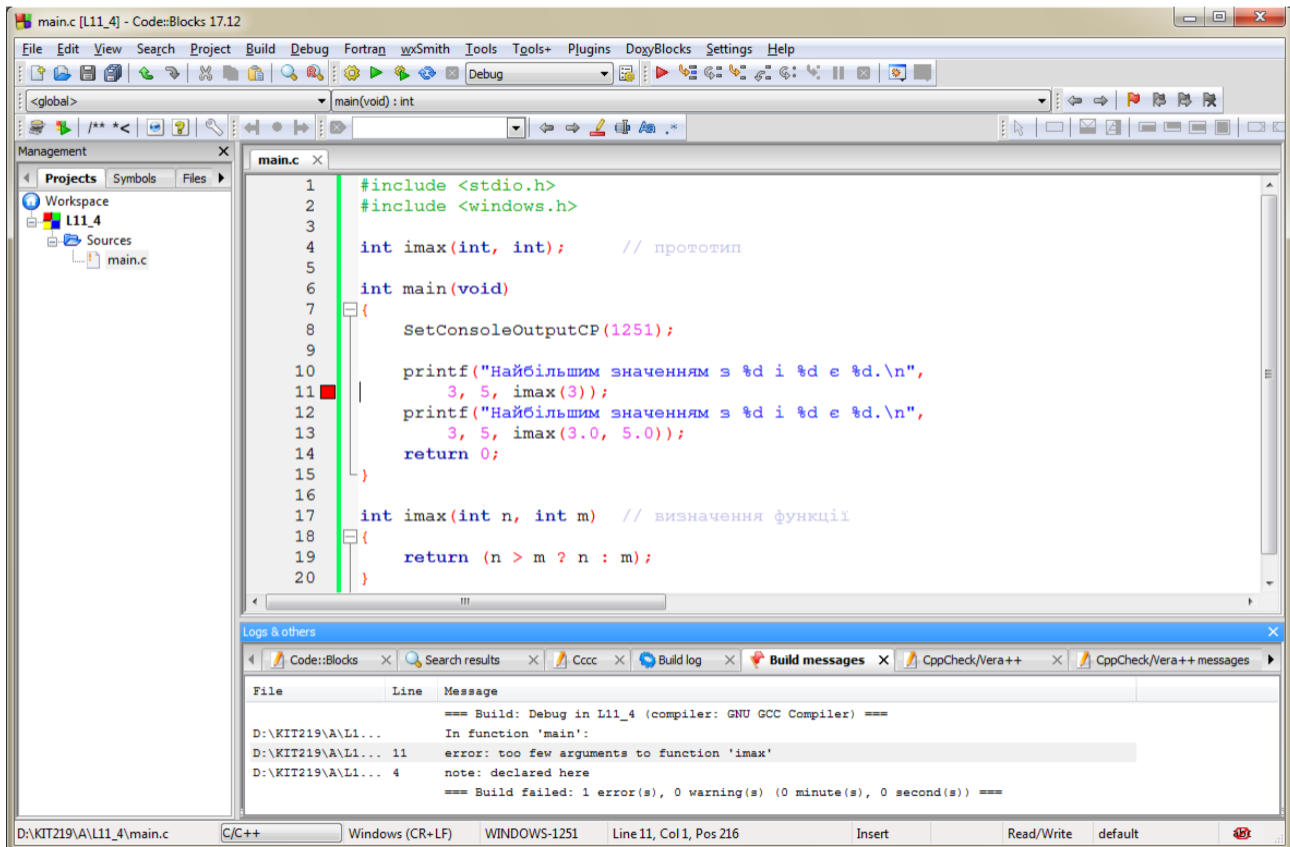


Рисунок 6.5 – Результат виконання програми для виявлення максимального значення з двох можливих

Як і очікувалося, `3.0` і `5.0` у другому виклику були перетворені на `3` і `5`, щоб функція могла необхідним чином обробити вхідні дані.

6.1.12. Відсутність аргументів і невизначені аргументи

Припустимо, що ви створили такий прототип:

```
void print_name();
```

Компілятор **ANSI C** припустить, що ви вирішили скористатися стилем оголошення, що передує зазначенню прототипу, й не буде перевіряти аргументи. Для позначення того, що функція не приймає аргументів, слід

вказати в круглих дужках ключове слово void:

```
void print_name(void);
```

Компілятор ANSI C інтерпретує попередній вираз таким чином, що функція print_name() не має аргументів. Потім він перевіряє, чи не застосовуєте ви аргументи під час виклику цієї функції.

Деякі функції, такі як printf() і scanf(), приймають змінну кількість аргументів. Наприклад, у printf() першим аргументом є рядок, але решта аргументів не фіксовані ні за типом, ні за кількістю. Для таких випадків стандарт ANSI C дозволяє часткове зазначення прототипу. Наприклад, для printf() можна використовувати такий прототип:

```
int printf(const char *,   ...);
```

Цей прототип вказує, що першим аргументом є рядок і що можуть бути вказані подальші аргументи з невизначеною природою.

Файл заголовку stdarg.h у бібліотеці функцій C надає стандартний спосіб для визначення функції зі змінною кількістю параметрів.

6.1.13. Перевага прототипів

Прототипи є потужним доповненням до мови C. Вони дозволяють компілятору виявляти більшість помилок, які могли бути зроблені під час використання функцій. Якщо їх не виявити своєчасно, вони перетворяться на проблеми, які можуть виявитися складними для відстеження. Чи зобов'язані ви використовувати прототипи? Ні, ви цілком можете використовувати старий метод оголошення функцій (без зазначення параметрів), але жодних переваг він не має, натомість маючи багато недоліків.

Існує один спосіб уникнути прототипу, одночасно зберігаючи переваги зазначення прототипу. Причиною зазначення прототипу є повідомлення компілятору про те, як повинна використовуватися функція, до досягнення їм першого фактичного випадку її застосування. Такого самого результату можна досягти, помістивши повне визначення функції до її першого

використання. В цьому випадку визначення діє як власний прототип. Частіше за все це робиться з короткими функціями:

```
// наступний код є визначенням і прототипом
int imax(int a, int b) { return a > b ? a : b; }

int main(void)
{
    int x, z;
    ...
z = imax(x, 50);
 ...
}
```

6.1.14. Рекурсія

В мові **C** функції дозволено викликати саму себе. Цей процес має назву **рекурсії**. Часом рекурсія буває складною, але іноді й дуже зручною. Складність пов'язана з доведенням рекурсії до кінця, оскільки функція, яка викликає сама себе, має тенденцію робити це нескінченно, якщо в коді не передбачена перевірка умови завершення рекурсії.

Рекурсія часто може застосовуватися там, де застосовується цикл. Іноді більш очевидним є рішення з циклом, а іноді – рішення з рекурсією. Рекурсивні рішення є більш елегантними, але менш ефективними, ніж рішення з циклами.

Використання рекурсії. Давайте поглянемо на приклад рекурсії. Текст програми має такий вид:

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

int main(void)
{

    SetConsoleOutputCP(1251);
    up_and_down(1);
    return 0;

}
void up_and_down(int n)
{
    printf("Рівень %d: комірка n %p\n", n, &n);    // 1

    if(n < 4)
        up_and_down(n + 1);
```

```
    printf("Рівень %d: комірка n %p\n", n, &n);    // 2
}
```

Результат роботи програми наведено на рис. 6.6.

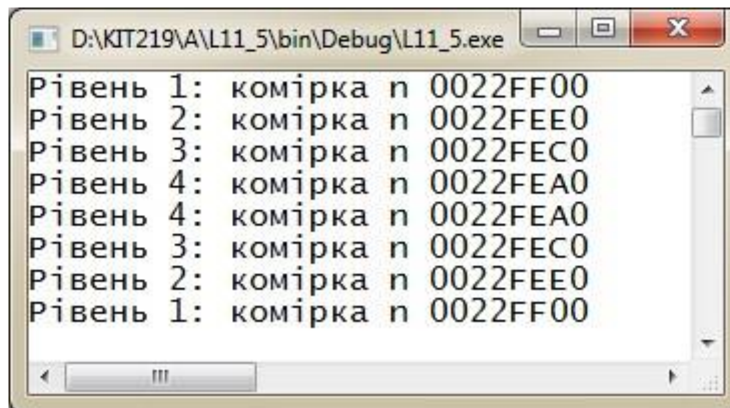


Рисунок 6.6 – Результат виконання програми з рекурсією

Функція `main()` в розглянутій програмі викликає функцію `up_and_down()`. Назвемо це «першим рівнем рекурсії». Потім функція `up_and_down()` викликає саму себе. Назвемо це «другим рівнем рекурсії». Другий рівень викликає третій рівень рекурсії і т. д. В цьому прикладі налаштовані чотири рівня рекурсії. Щоб поглянути, що відбувається всередині, програма не тільки відображає значення змінної `n`, але також і значення `&n`, яке являє собою адресу комірки пам'яті, де зберігається змінна `n`.

Давайте переглянемо цю програму, щоб зрозуміти, як працює рекурсія. Спочатку `main()` викликає `up_and_down()` з аргументом `1`. В результаті формальний параметр `n` функції `up_and_down()` отримує значення `1`, тому перший оператор виводу відображає рядок `Рівень 1:`. Далі, оскільки `n` менше `4`, функція `up_and_down()` (`рівень 1`) викликає функцію `up_and_down()` (`рівень 2`) з фактичним аргументом `n + 1`, або `2`. Це призводить до того, що `n` у виклику `рівня 2` присвоюється значення `2`, так що перший оператор виводу відображає рядок `Рівень 2:`. Аналогічно, наступні два виклики призводять до виводу `Рівень 3:` і `Рівень 4:`.

Коли буде досягнутий `рівень 4`, змінна `n` дорівнює `4`, тому перевірка

в **if** не відбувається. Функція **up_and_down()** не викликається знову. Замість цього виклик **рівня 4** продовжується виконанням другого оператора виводу, який відображає рядок **Рівень 4:**, оскільки змінна **n** має значення **4**. В цій точці виклик **рівня 4** закінчується, а керування повертається функції, яка його ініціювала (виклик **рівня 3**). Останнім оператором, який виконався всередині виклику **рівня 3**, був виклик **рівня 4** в операторі **if**. Отже, виконання рівня 3 поновлюється з наступного оператора, яким є другий оператор виводу. Це призводить до відображення рядка **Рівень 3:**. Потім **рівень 3** завершується, передаючі керування **рівню 2**, який виводить рядок **Рівень 2:**, і т. д.

Зверніть увагу, що на кожному рівні рекурсії застосовується власна закрита змінна **n**. Цей факт легко встановити, якщо поглянути на значення адрес.

Якщо ви вважаєте це пояснення дещо заплутаним, то уявіть, що є ланцюжок викликів функцій, в якому **fun1()** викликає **fun2()**, **fun2()** викликає **fun3()** і **fun3()** викликає **fun4()**. Коли **fun4()** завершується, керування передається **fun3()**. По завершенні **fun3()** керування передається **fun2()**. Коли закінчується **fun2()**, керування повертається знову до **fun1()**. Рекурсивний варіант працює так само, за виключенням того, що функції **fun1()**, **fun2()**, **fun3()** і **fun4()** є однією і тією ж самою функцією.

Основи рекурсії. Спочатку рекурсія може бути незрозумілою, тому давайте розглянемо декілька базових аспектів, які допоможуть зрозуміти процес.

По-перше, кожен рівень виклику функції має власні змінні. Тобто змінна **n рівня 1** відрізняється від змінної **n рівня 2**. Таким чином програма створить чотири різних змінних, кожна з яких має ім’я **n** і власне значення, яке відрізняється від інших. Коли в решті решт програма повертається до виклику функції **up_and_down() рівня 1**, початкова змінна **n** як і раніше має значення **1**, з якого вона починала (рис. 6.7).

Змінні:	**n n n n**
Після виклику рівня 1	1
Після виклику рівня 2	1 2
Після виклику рівня 3	1 2 3
Після виклику рівня 4	12 3 4
Після повернення з рівня 4	1 2 3
Після повернення з рівня 3	1 2
Після повернення з рівня 2	1
Після повернення з рівня 1	

(все завершено)

Рисунок 6.7 – Змінні рекурсії

По-друге, кожен виклик функції збалансований з поверненням. Коли потік керування досягає оператора `return` наприкінці останнього рівня рекурсії, керування переходить на попередній рівень рекурсії.

Перехід одразу до початкового виклику всередині `main()` не відбувається. Замість цього керування повинно пройти через кожен рівень рекурсії, повертаючись з одного рівня `up_and_down()` на рівень функції `up_and_down()`, яка її викликала.

По-третє, оператори в рекурсивній функції, які передують рекурсивному виклику, виконуються в тому ж самому порядку, в якому ці функції викликалися. Наприклад, в розглянутій програмі перший оператор виводу знаходиться попереду рекурсивного виклику. Він був виконаний чотири рази в порядку надходження рекурсивних викликів: `рівень 1`, `рівень 2`, `рівень 3` і `рівень 4`.

По-четверте, оператори в рекурсивній функції, які знаходяться після рекурсивного виклику, виконуються в порядку, зворотному тому, в якому ці функції викликалися. Наприклад, другий оператор виводу розміщується після рекурсивного виклику, і він виконується в наступному порядку: `рівень 4`, `рівень 3`, `рівень 2`, `рівень 1`. Ця властивість рекурсії є корисною при програмуванні задач, що передбачають зміну порядку на

протилежний.

По-п'яте, хоча кожен рівень рекурсії має власний набір змінних, сам код не дублюється. Код – це послідовність інструкцій, а виклик функції являє собою команду переходу на початок цієї послідовності інструкцій. Рекурсивний виклик потім повертає програму на початок зазначеної послідовності інструкцій. Якщо не звертати увагу на те, що рекурсивні виклики створюють нові змінні при кожному виклику, вони дуже нагадують цикл. Насправді часом рекурсія може бути використана замість циклу та навпаки.

Нарешті, по-шосте, дуже важливо, щоб рекурсивна функція містила код, який міг би зупинити послідовність рекурсивних викликів. Зазвичай в рекурсивній функції застосовується перевірка **if** або її еквівалент для припинення рекурсії, коли будь-який параметр функції досягає певного значення. Щоб це працювало, в кожному виклику повинне використовуватися значення, яке буде відмінним для цього параметра. В останньому прикладі функція **up_and_down(n)** викликає **up_and_down(n+1)**. У підсумку фактичний аргумент досягає значення **4** і перевірка умови **if(n < 4)** не проходить.

Хвостова рекурсія . В найпростішій формі рекурсії рекурсивний виклик знаходиться наприкінці функції, безпосередньо попереду оператора **return**. Така рекурсія зветься **хвостовою** або **кінцевою**, тому що рекурсивний виклик відбувається наприкінці. Хвостова рекурсія є найпростішою формою, оскільки вона діє подібно циклу.

Давайте розглянемо версії з циклом і хвостовою рекурсією для функції обчислення факторіалу. Факторіал цілого числа – це результат добутку усіх цілих чисел, починаючи з **1** і закінчуючи заданим числом. Наприклад, факторіал **3** (записується як **3!**) відповідає добутку **1·2·3**. Крім того, **0!** приймається рівним **1**, а **для від'ємних чисел факторіали не визначені**. Розглянемо програму, де в одній функції для обчислення факторіалу застосовується цикл **for**, а в другій функції – рекурсія. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
long fact(int n);
long rfact(int n);
int main(void)
{
   int num;
   SetConsoleOutputCP(1251);
   printf("==============================================\n");
   printf(" Програма для обчислення факторіалу          \n");
   printf("==============================================\n");
   printf("Введіть значення в діапазоні 0-12"
          " (q для завершення):\n");
   while(scanf("%d", &num) == 1)
  {
        if(num < 0)
            printf("Від\'ємні числа не підходять.\n");
        else
        {
            if(num > 12) {
                printf("Значення, що вводиться, повинно" "
                       бути менше 13.\n");
        }
            else
            {
                printf("\nЦикл:     %d! = %ld\n",
                       num, fact(num));
                printf("Рекурсія: %d! = %ld\n",
                       num, rfact(num));
            }
        }
        printf("\nВведіть значення в діапазоні 0-12"
               " (q для завершення):\n");
  }
    printf("Програма завершена.\n");
    return 0;
}
long fact(int n)    // функція, що основана на циклі
{
    long ans;

    for(ans = 1; n > 1; n--)
        ans *= n;
    return ans;
}
long rfact(int n)   // рекурсивна версія
{
    long ans;

    if(n > 0)
        ans = n * rfact(n - 1);
```

```
        else
            ans = 1;
        return ans;
    }
```

Результат виконання програми для обчислення факторіала за допомогою функції `fact()`, яка містить цикл, і рекурсивної функції `rfact()`, наведено на рис. 6.8.

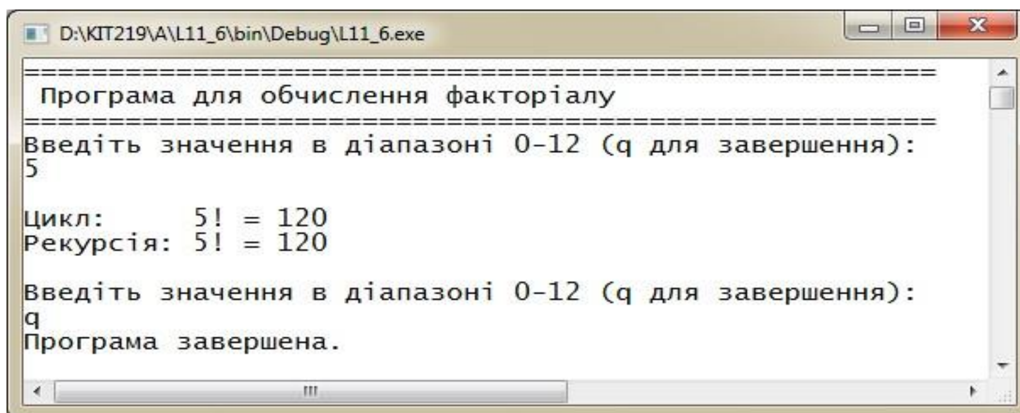


Рисунок 6.8 – Результат виконання програми для обчислення факторіала

Програма тестового драйвера обмежує вхідні дані цілими значеннями в діапазоні від `0` до `12`. Виявляється, що значення `12!` є дещо меншим за півмільярда, тому результат `13!` займає значно більше пам’яті в комп’ютері, ніж тип `long`. Для обчислення факторіалів, що перевищують `12!`, доведеться використовувати тип більшого розміру, такий як `double` або `long long`.

Версія з циклом ініціалізує змінну `ans` значенням `1`, а потім помножує її на цілі числа від `n` до `2`. Формально треба було б помножити також і на `1`, але результат від цього не зміниться.

Тепер поглянемо на рекурсивну версію. Ключовим моментом є рівняння `n!=n·(n-1)!`. Воно випливає з того факта, що `(n-1)!` є добутком усіх додатних цілих чисел до . Таким чином, множення його на `n` дає добуток цілих чисел аж до `n`. Це добре вписується в рекурсивний підхід. Якщо назвати функцію `rfact()`, то `rfact(n)` відповідає `n* rfact(n-1)`. Отже, обчислити значення `rfact(n)` можна, зробивши виклик в ній

rfact(n-1**)**, як це робиться в програмі. Зрозуміло, необхідно перервати рекурсію в будь-якій точці, і це можна зробити, встановивши значення, що повертається, в **1**, коли дорівнює **0**.

Рекурсивна версія програми дає той самий результат, що й версія з циклом. Зверніть увагу, що хоча виклик **rfact()** не є останнім рядком у функції, це останній оператор, який виконується, коли n>0, тобто ми маємо справу з **хвостовою рекурсією**.

Враховуючи можливість застосування в коді функції або циклу, або рекурсії, якому підходу треба надавати перевагу? Зазвичай цикл є більш вдалим вибором. По-перше, через те, що кожен рекурсивний виклик створює власний набір змінних, варіант з рекурсією використовує більше пам'яті; кожен рекурсивний виклик поміщає в стек новий набір змінних. При цьому обмежений об'єм стеку може встановлювати межу кількості рекурсивних викликів. По-друге, рекурсія виконується повільніше, оскільки кожен виклик функції займає певний час. Для чого тоді потрібен цей приклад? Причина в тому, що хвостова рекурсія є найпростішою формою рекурсії для її розуміння, а рекурсія заслуговує освоєння, оскільки в ряді випадків проста альтернатива у вигляді циклу відсутня.

Рекурсія та зміна порядку на протилежний. Давайте тепер розглянемо задачу, для якої здатність рекурсії змінювати порядок на протилежний виявляється корисною. Це саме той випадок, коли рекурсія є простішою, ніж використання циклу. Задача полягає в написанні функції, яка виводить двійковий еквівалент цілого числа.

У двійковому запису числа представляються степенями **2**. Подібно тому, як **234** в десятковому вигляді означає $2\times10^{2} + 3\times10^{1} + 4\times10^{0}$, число **101** в двійковому вигляді представляється як $1\times2^{2} + 0\times2^{1} + 1\times2^{0}$. В двійкових числах використовуються тільки цифри **0** і **1**.

Для вирішення задачі потрібен деякий метод, або алгоритм. Наприклад, яким чином можна знайти двійковий еквівалент **5**? Ясно, що непарні числа

повинні мати двійкове представлення, яке закінчується цифрою **1**. Парні числа закінчуються цифрою **0**, тому можна визначити, чи є остання цифра **1** або **0**, обчисливши значення **5 % 2**. Якщо результат дорівнює **1**, то число **5** – непарне, і останньою цифрою буде **1**. В загальному випадку, якщо **n** – число, то останньою цифрою буде **n % 2**, тому перша знайдена цифра – це остання цифра, яку треба вивести. Це передбачає застосування рекурсивної функції, в якій вираз **n % 2** обчислюється до рекурсивного виклику, але результат виводиться після нього. Таким чином, перше обчислене значення є останнім значенням, що виводиться.

Щоб отримати наступну цифру, треба поділити початкове число на **2**. Це двійковий еквівалент зсуву десяткової крапки на одну позицію вліво, що дозволить з'ясувати наступну двійкову цифру. Якщо отримаємо парне значення, то наступною двійковою цифрою буде **0**, а якщо непарне – то **1**. Наприклад, **5/2** дає **2** (цілочислове ділення). Таким чином, наступна цифра – **0**. Тепер маємо **01**. Далі повторимо цей процес, поділивши **2** на **2**, щоб отримати **1**. Обчислення **1 % 2** дає **1**, тому наступною цифрою буде **1**. В результаті маємо **101**. Коли необхідно зупинитися? Очевидно тоді, коли результат ділення на **2** виявляється менше **2**, оскільки поки він залишається рівним **2** або більше, існує ще одна двійкова цифра. Кожне ділення на **2** скорочує на одну двійкову цифру, поки не буде досягнутий кінець.

Текст програми з застосуванням рекурсії, яка змінює порядок на протилежний має такий вигляд:

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

void to_binary(unsigned long n);

int main(void)
{
   unsigned long number;

   SetConsoleOutputCP(1251);

   printf("Введіть ціле число (q для завершення):\n");
   while(scanf("%lu", &number) == 1)
```

```
   {
       printf("Двійковий еквівалент: ");
       to_binary(number);
       putchar('\n');
       printf("Введіть ціле число (q для завершення):\n");

   }
   printf("Програма завершена.\n");
   return 0;
}

void to_binary(unsigned long n)          // рекурсивна функція
{
    int r;

    r = n % 2;
    if(n >= 2)
         to_binary(n / 2);
         putchar(r == 0 ? '0' : '1');
         return;
}
```

Функція `to_binary()` повинна відображати символ `'0'`, якщо числове значення змінної **`r`** дорівнює `0`, і `'1'`, якщо воно дорівнює `1`. Умовний вираз **`r == 0 ? '0' : '1'`** забезпечує таке перетворення числових значень на символи.

Результат виконання програми наведено на рис. 6.9.

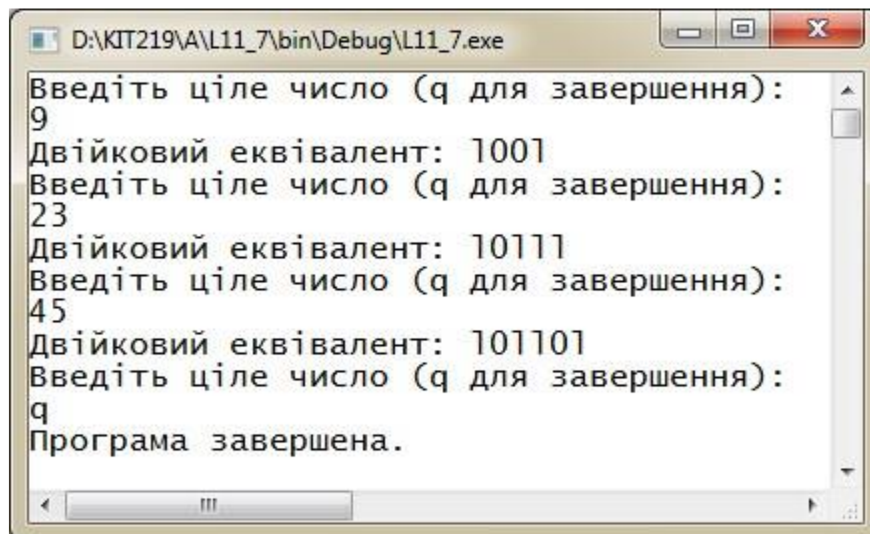


Рисунок 6.9 – Результат виконання програми для переведення числа з десяткової у двійкову систему числення

Чи можна скористатися цим алгоритмом для обчислення двійкового представлення числа без застосування рекурсії? Так, це можливо. Але оскільки даний алгоритм першою обчислює останню цифру, то перш ніж відобразити результат, всі цифри прийшлось би десь зберігати (наприклад, в масиві).

Рекурсія має як свої переваги так і недоліки. Однією з переваг рекурсії є те, що вона пропонує найпростіше рішення ряду задач програмування. Один з недоліків полягає в тому, що деякі рекурсивні алгоритми можуть швидко вичерпати ресурси пам'яті комп'ютера. Крім того, рекурсію важко документувати та супроводжувати. Розглянемо приклад, який ілюструє як переваги так і недоліки рекурсії.

Числа Фібоначчі можна визначити наступним чином: перше число Фібоначчі – це **1**, друге число Фібоначчі – також **1**, а кожне наступне число Фібоначчі є сумою двох попередніх чисел. Таким чином, перші декілька чисел в послідовності виглядають так: **1, 1, 2, 3, 5, 8, 13**. Створимо функцію, яка для заданого додатного цілого числа **n** повертає відповідне число Фібоначчі.

Спочатку відзначимо перевагу рекурсії: вона забезпечує просте визначення. Якщо назвати функцію **Fibonacci()**, то **Fibonacci(**n**)** повинна повертати значення **1**, якщо **n** дорівнює **1** або **2**, і суму **Fibonacci(**n-1**)** і **Fibonacci(**n-2**)** в іншому випадку:

```
unsigned long Fibonacci(unsigned n)
{
    if(n > 2)
         return Fibonacci(n-1) + Fibonacci(n-2);
    else
         return 1;
}
```

Рекурсивна функція **C** просто повторює математичне визначення рекурсії. В цій функції використовується подвійна рекурсія, тобто викликає себе двічі. Ця обставина є джерелом її слабкості.

Щоб побачити природу цієї слабкості, припустимо, що є виклик `Fibonacci(40)`. Це буде перший рівень рекурсії, і він виділяє пам'ять для змінної `n`. Потім він викликає функцію `Fibonacci()` два рази, створюючи на другому рівні рекурсії ще дві змінні `n`. Кожен з цих двох викликів генерує ще два виклики, які, у свою чергу, потребують ще чотирьох змінних з іменами `n` на третьому рівні рекурсії, що в сумі дає сім змінних. На кожному рівні кількість змінних подвоюється у порівнянні з попереднім рівнем, тобто об'єм змінних зростає за експонентою! Експоненціальне зростання швидко призводить до величезних значень. В нашому випадку експоненціальне зростання швидко призведе до того, що комп'ютеру буде потрібен гігантський об'єм пам'яті. Це, скоріш за все, призведе до аварійного завершення програми.

Насправді це екстремальний приклад, однак він добреілюструє необхідність у дотриманні обережності під час застосування рекурсії, особливо коли важливим фактором є ефективність.

6.2. Вказівники

Вказівники є однією з самих потужних особливостей мови програмування С. Разом з цим, вони є однією з самих складних тем. Використання вказівників дозволяє реалізувати передачу аргументів функціям за посиланням, передачу функцій між функціями, а також створювати динамічні структури даних і керувати ними (які можуть збільшуватися або зменшуватися в розмірах під час виконання), такі як зв'язані списки, черги, стеки та дерева. Сьогодні ми розглянемо основні поняття, які пов'язані з вказівниками. В подальшому ми будемо досліджувати застосування вказівників для роботи зі структурами, описувати прийоми динамічного керування пам'яттю, створенням і застосуванням динамічних структур даних.

6.2.1. Змінні-вказівники, визначення та ініціалізація

Вказівники – це змінні, значеннями яких є адреси в пам'яті. Звичайні змінні містять безпосередні значення. Вказівник, навпаки, містить адресу змінної, яка зберігає безпосереднє значення. В цьому сенсі ім'я змінної є безпосереднім посиланням на значення, а вказівник – непрямим (рис. 6.10). Доступ до значення за допомогою вказівника має назву непрямого доступу.

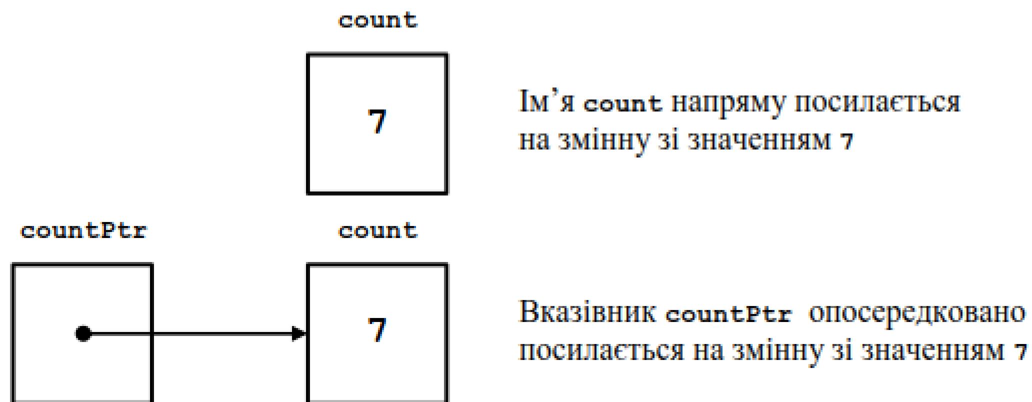


Рисунок 6.10 – Прямі та непрямі посилання на значення

Оголошення вказівників. Вказівники, як і звичайні змінні, повинні визначатися перш ніж їх можна буде використовувати. Наступне визначення:

```
int *countPtr, count;
```

оголошує змінну типу **int *** (тобто вказівник на ціле число) і читається як «**countPtr** – вказівник на значення типу **int**» або «**countPtr** вказує на об'єкт типу **int**». Тут також оголошується змінна **count** типу **int**, а не вказівник на значення типу **int**. Специфікатор ***** застосовується тільки до імені **countPtr** у визначенні. Коли символ ***** використовується у визначенні таким способом, він повідомляє, що змінна, яка визначається, є вказівником. Вказівники можуть вказувати на об'єкти будь-яких типів. Щоб виключити неоднозначність при визначенні змінних-вказівників і звичайних змінних в одній інструкції, в одному оголошенні завжди слід визначати тільки одну змінну.

Ініціалізація вказівників і присвоювання їм значень. Вказівники можуть

ініціалізуватися в момент оголошення, а також їм можна присвоювати значення вже в ході виконання. Вказівник може бути ініціалізований значенням **NULL**, **0** або адресою. Вказівник зі значенням **NULL** вказує в нікуди, **NULL** – це символічна константа, яка оголошена у файлі заголовку **stddef.h** (її оголошення також можна знайти в деяких інших файлах заголовку, таких як **stdio.h**). Ініціалізація вказівника значенням **0** еквівалентна ініціалізації значенням **NULL**, але взагалі краще використовувати **NULL**. Коли вказівнику присвоюється значення 0, воно спочатку перетворюється на вказівник відповідного типу. Значення **0** – єдине цілочислове значення, яке може бути присвоєно змінній-вказівнику безпосередньо.

6.2.2. Оператори вказівників

Амперсанд (**&**), або оператор взяття адресу, – це унарний оператор, що повертає адресу свого операнда. Наприклад, припустимо, що є наступні визначення:

```
int y = 5;
int *yPtr;
```

Тоді інструкція

```
yPtr = &y;
```

присвоїть змінній-вказівнику **yPtr** адресу змінної **y**. Після цього змінній **yPtr** можна сказати, що вона «вказує на» змінну **y**. На рис. 6.11 показано схематичне представлення пам’яті після виконання присвоювання.

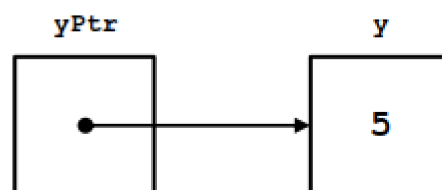


Рисунок 6.11 – Графічне представлення вказівника, що вказує на цілочислову змінну

Представлення вказівника в пам'яті. На рис. 6.12 показано представлення вказівника в пам’яті, де передбачається, що цілочислова змінна

y зберігається в пам'яті за адресою **600000**, а змінна-вказівник **yPtr** – за адресою **500000**. Операнд оператора взяття адреси обов'язково повинен бути змінною. Цей оператор не може застосовуватися до констант і виразів.

Рисунок 6.12 – Представлення змінних **y** і **yPtr** в пам'яті

Оператор непрямого звернення (*). Унарний оператор *, часто має назву оператора непрямого звернення або оператора розіменування, повертає значення об'єкта, на який вказує його операнд (тобто вказівник). Наприклад, інструкція

```
printf("%d", *yPtr);
```

виведе значення змінної **y**, а саме число **5**. Такий спосіб використання унарного оператора * має назву розіменування вказівника.

Розіменування неініціалізованого вказівника або вказівника, якому не була присвоєна дійсна адреса змінної в пам'яті, є помилкою. Це може привести до аварійного завершення програми або до пошкодження важливих даних і отриманню помилкових результатів.

Демонстрація операторів & і *. В наступній програмі демонструється використання операторів вказівників **&** і *. Текст програми має такий вигляд:

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

int main(void)
{
  int a;              // a - цілочислова змінна
  int *aPtr;          // aPtr - вказівник на цілочислову змінну

  SetConsoleOutputCP(1251);
```

```
a = 7;
aPtr = &a;          // присвоїти змінній aPtr адресу змінної a

printf("Адреса змінної a:              %p\n"
       "Адреса змінної aPtr:           %p", &a, aPtr);
printf("\n\nЗначення змінної a:    %d\n"
       "Значення змінної *aPtr         %d", a, *aPtr);
printf("\n\nПоказуємо, що оператори * і & доповнюють"
       " один одного:\n\n&*aPtr = %p\n"
       "*&aPtr = %p\n", &*aPtr, *&aPtr); return 0; }
```

Результат виконання програми наведено на рис. 6.13.

Специфікатор перетворення %p використовується у виклику функції `printf()` для виводу адресів у пам'яті в шістнадцятковому вигляді. Зверніть увагу, що адреса змінної **a** співпадає зі значенням змінної **aPtr**, це підтверджує, що змінній **aPtr** дійсно присвоєна адреса. Оператори **&** и ***** доповнюють один одного. Коли вони обидва послідовно застосовуються до змінної **aPtr** в будь-якому порядку, виводиться однаковий результат.

```
D:\KIT219\A\L12_1\bin\Debug\L12_1.exe

Адреса змінної a:        0022FF0C
Адреса змінної aPtr:     0022FF0C

Значення змінної a:      7
Значення змінної *aPtr   7

Показуємо, що оператори * і & доповнюють один одного:

&*aPtr = 0022FF0C
*&aPtr = 0022FF0C
```

Рисунок 6.13 – Приклад використання операторів вказівників **&** і *****

6.2.3. Передача аргументів функціям за посиланням

Існує два способи передачі аргументів функціям: за значенням і за посиланням. Усі аргументи в мові **C** передаються за значенням. Як було показано раніше, для повернення значення (або повернення керування без повернення значення) функцією, яка викликається, до функції, яка її викликала, можна використовувати інструкцію **return**. Часто необхідно мати можливість модифікувати змінні, які оголошені в функції, яка викликає,

або передавати вказівник на великий об'єкт даних, щоб уникнути накладних витрат на копіювання цього об'єкта при передачі за значенням (що вимагає часу і пам'яті для зберігання копії об'єкта).

У мові **C** вказівники та оператор непрямого доступу використовуються для імітації передачі аргументів за посиланням. Коли функція повинна змінити значення аргументу, їй передається адреса цього аргументу. Зазвичай це реалізується застосуванням оператора взяття адреси (**&**) до змінної (яка оголошена в функції, що викликає), значення якої повинно бути змінено. Масиви передаються функцій без застосування оператора **&**, тому що компілятор мови **C** автоматично передає масив як адресу першого його елемента (ім'я масиву діє еквівалентно виразу **&arrayName[0]**). Коли функції передається адреса змінної, всередині неї можна використовувати оператор непрямого доступу (*****), щоб змінити значення, що зберігається за цією адресою.

Передача за значенням. Розглянемо програму, що викликає функцію **cubeByValue()**, яка підносить вказане цілочислове значення до третьої степені. В ній змінна передається функції **cubeByValue()** за значенням. Ця функція підносить значення аргументу до третьої степені та повертає отриманий результат за допомогою інструкції **return**. Нове значення потім присвоюється змінній **number**, яка оголошується в функції **main()**. Текст програми має такий вигляд:

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

int cubeByValue(int n);   // прототип функції

int main(void)
{
    int number = 5;        // ініціалізувати змінну number

    SetConsoleOutputCP(1251);

    printf("Початкове значення змінної number: %d", number);
    // передати number функції cubeByValue() по значенню
    number = cubeByValue(number);
    printf("\nНове значення змінної value: %d\n", number);
    return 0;
}
```

```
    int cubeByValue(int n)
    {
        return n * n * n; // піднести число до третьої степені
                          // та повернути результат
    }
```

Результат виконання програми наведено на рис. 6.14.

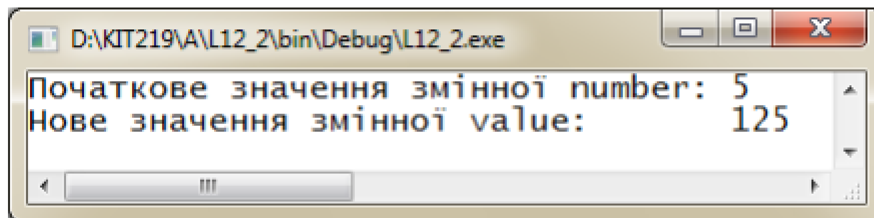


Рисунок.6.14 – Приклад передачі аргументу функції за значенням

Передача за посиланням. В наступній програмі змінна **number** передається за посиланням – функції **cubeByReference()** передається адреса змінної **number**. Функція **cubeByReference()** приймає в якості параметру вказівник на ціле число з ім'ям **nPtr**. Вона розіменовує вказівник, підносить до третьої степені число, на яке вказує змінна **nPtr**, і присвоює результат ***nPtr** (фактично це змінна **number** в функції **main()**), модифікуючи тим самим значення змінної **number**.

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

void cubeByReference(int *nPtr);    // прототип функції

int main(void)
{
    int number = 5;            // ініціалізувати змінну number

    SetConsoleOutputCP(1251);

    printf("Початкове значення змінної number: %d", number);
    // передати функції cubeByReference() адресу
    // змінної number
    cubeByReference(&number);
    printf("\nНове значення змінної value:         %d\n", number);
    return 0;
}

// підносить *nPtr до третьої степені
```

```
// фактично змінює number

void cubeByReference(int *nPtr)
{
     // піднести *nPtr до третьої степені
     *nPtr = *nPtr * *nPtr * *nPtr;
}
```

Результат виконання програми наведено на рис. 6.15.

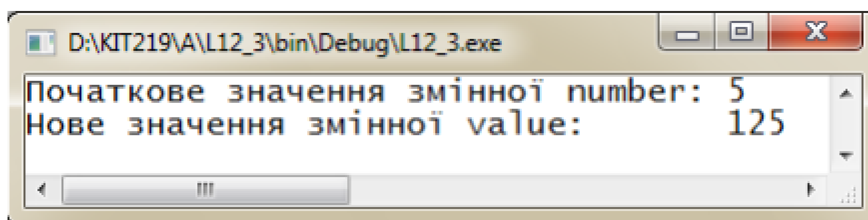


Рисунок 6.15 – Приклад передачі аргументу функції за посиланням

Функція, що приймає адресу змінної у вигляді аргументу, повинна визначити параметр-вказівник. Наприклад, у попередній програмі заголовок

функції `cubeByReference()` має такий вигляд:

```
void cubeByReference(int *nPtr)
```

Даний заголовок вказує на те, що `cubeByReference()` приймає адресу цілочислової змінної, зберігає цю адресу у своїй локальній змінній **`nPtr`** і нічого не повертає.

Прототип функції `cubeByReference()` містить оголошення `int *` в круглих дужках. Як і у випадках з параметрами інших типів, ім’я параметра-вказівника в прототипі функції можна опустити. Імена були включені виключно для документування коду – вони ігноруються компілятором.

Функції, які використовують одновимірні масиви як аргументи. Якщо функція приймає в якості аргументу одновимірний масив, в списку параметрів у прототипі та в заголовку функції можна використовувати форму запису вказівників, як в заголовку функції `cubeByReference()`. Компілятор **C** не розрізняє ситуації, коли функції приймають вказівники та одновимірні масиви. Однак сама функція повинна «знати», що вона приймає – одновимірний

масив чи адресу єдиної змінної. Коли компілятор зустрічає визначення параметра у формі одновимірного масиву `int b[ ]`, він перетворює його у визначення параметра вказівника `int *b`. Ці дві форми є взаємопов'язані.

Використовуйте передачу аргументів за значення, якщо функції, яка викликає, явно не треба, щоб функція, яка викликається, змінювала значення змінної, яка передається. Цей прийом стане запобіжником для випадкової зміни адреси і ще одним прикладом використання принципу найменших привілеїв.

6.2.4. Використання кваліфікатора constз вказівниками

Кваліфікатор `const` дає можливість повідомити компілятор про те, що значення позначеної ним змінної не повинне змінюватися.

Кваліфікатор `const` можна використовувати для примусового проведення у життя принципу найменших привілеїв. Він допоможе скоротити час на відлагодження програми, усунути небажані побічні ефекти та спростити супровід і подальший розвиток програми.

За довгі роки був накопичений великий об'єм коду, який написаний на ранніх версіях мови `C`, де була відсутня підтримка кваліфікатора `const`. В наслідок цього з'явилась маса можливостей по вдосконаленню та реорганізації старого коду на `C`.

Застосування `const` до параметрів функцій. Існує шість варіантів застосування (або незастосування) кваліфікатора до параметрів функцій – два варіанти до параметрів, які передаються за значенням, і чотири варіанти до параметрів, які передаються за посиланням. Як правильно обрати той чи інший варіант в кожному конкретному випадку? Треба дотримуватися принципом найменших привілеїв.

Завжди треба надавати функціям доступ до даних за допомогою параметрів, який буде достатнім для рішення поставленої задачі, але не більше.

Раніше ви вже дізналися, що в мові `C` все аргументи передаються функціям за значенням, тобто функції отримують копії параметрів. якщо

функція змінить копію, це жодним чином не вплине на оригінальну змінну, яка міститься у функції, яка викликає. В багатьох випадках функції змінюють свої аргументи в процесі рішення задачі. Однак в деяких ситуаціях значення не повинно змінюватися у функції, яка викликає, навіть тоді, коли аргумент є лише копією оригінального значення.

Уявіть собі функцію, яка приймає у вигляді аргументів одновимірний масив і його розмір та виводить вміст масиву на екран. Така функція повинна в циклі обійти елементи масиву та вивести їх на консоль. Аргумент з розміром масиву використовується в тілі функції для визначення максимально допустимого індексу, щоб цикл міг завершитися після виводу останнього елемента. Ні розмір, ні вміст масиву не повинні змінюватися в тілі функції.

Якщо значення деякого аргументу не змінюється (або не повинне змінюватися) в тілі функції, відповідний параметр треба оголосити з кваліфікатором `const`, щоб попередити випадкову його зміну.

Якщо в функції буде виявлена спроба змінити значення аргументу, який був оголошений з кваліфікатором `const`, компілятор виявить її і виведе або попередження, або повідомить про помилку, в залежності від версії компілятора.

Намагання передати аргумент за значенням функції, що приймає вказівник, є помилкою. Деякі компілятори передають значення, припускаючи, що вони є вказівниками і виконують розіменування. Під час виконання це часто призводить до помилок порушення прав доступу до пам'яті. Інші компілятори виявляють невідповідність типів аргументів і параметрів і генерують повідомлення про помилки під час компіляції.

Існує чотири способи передачі вказівників до функції:

1) передача вказівника, що змінюється, на дані, що змінюються;

2) передача константного вказівника на дані, що змінюються;

3) передача вказівника, що змінюється, на константні дані;

4) передача константного вказівника на константні дані.

Кожна з чотирьох комбінацій забезпечує різні привілеї доступу.

Перетворення рядків у верхній регістр з використанням вказівника, що змінюється, на дані, що змінюються. Найвищі привілеї доступу до даних видаються, коли функція отримує вказівник, що змінюється, на дані, що змінюються. В цьому випадку дані можна змінювати за допомогою розіменування вказівника, і сам вказівник можна змінювати з метою отримати доступ до інших елементів даних. Оголошення вказівника, що змінюється, на дані, що змінюються, відбувається без використання кваліфікатора `const`. Такий вказівник може використовуватися, наприклад, для передачі рядка функції, яка обробляє (і можливо, змінює) кожен символ в рядку.

Розглянемо приклад, в якому представлена функція `convertToUppercase()`, яка оголошує свій параметр `sPtr(char *sPtr)` як вказівник, що змінюється, на дані, що змінюється. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <ctype.h>
   void convertToUppercase(char *sPtr);      // прототип функції

   int main(void)
   {
       char string[] = "cHaRaCters and $32.98";

       SetConsoleOutputCP(1251);

       printf("Рядок до перетворення:    %s", string);
       convertToUppercase(string);
       printf("\nРядок після перетворення: %s\n", string);
       return 0;
   }

   // перетворює символи рядка у верхній регістр

   void convertToUppercase(char *sPtr)
   {
       while(*sPtr != '\0')
       {
          *sPtr = toupper(*sPtr);   // перетворити у верхній регістр
          ++sPtr;                   // перейти до наступного
          // перейти до наступного символу
       }
   }
```

Результат виконання програми наведено на рис. 6.16.

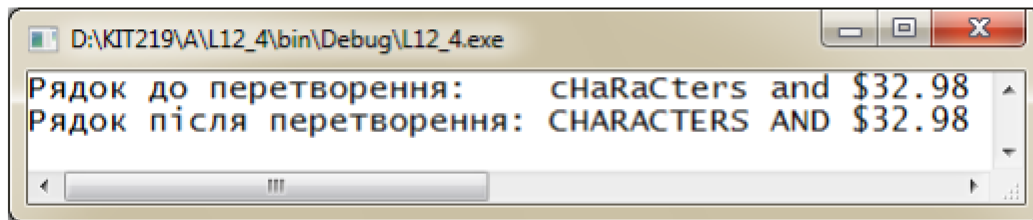


Рисунок 6.16 – Приклад передачі вказівника в якості аргументу функції

Функція обробляє масив **string** (на який вказує аргумент **sPtr**) по одному символу. Вона перетворює кожен символ у верхній регістр, викликаючи стандартну функцію **toupper()** з файлу заголовка **ctype.h**, – якщо поточний символ не є буквою або вже є буквою верхнього регістра, **toupper()** поверне цей символ. Інструкція `++sPtr;` переміщує вказівник до наступного символу.

Вивід рядка по одному символу з використанням вказівника, що змінюється, на константні дані. Вказівник, що змінюється, на константні дані можна змінити, зберігши в ньому адресу будь-якого іншого елемента даних відповідного типу, але ці дані, на які він вказує, змінити не можна. Такий вказівник можна використовувати для прийому аргументу-масиву в функції, яка обробляє кожен елемент масиву, але не змінює дані.

Розглянемо програму, в якій функція **printCharacters()** оголошує параметр **sPtr** типу **const char ***. Текст програми має такий вигляд:

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

void printCharacters(const char *sPtr);

int main(void)
{
      // ініціалізація масиву символів
      char string[] = "print characters of a string";
      SetConsoleOutputCP(1251);
      puts("Вхідний рядок:");
      printCharacters(string);
      puts("");
```

```
    return 0; }

// вказівник sPtr не може використовуватися
// для зміни символу, на який він вказує,
// тобто sPtr - вказівник "тільки для читання"

void printCharacters(const char *sPtr)
{
    for( ; *sPtr != '\0'; ++sPtr)
        printf("%c", *sPtr);
}
```

Результат виконання програми наведено на рис. 6.17.

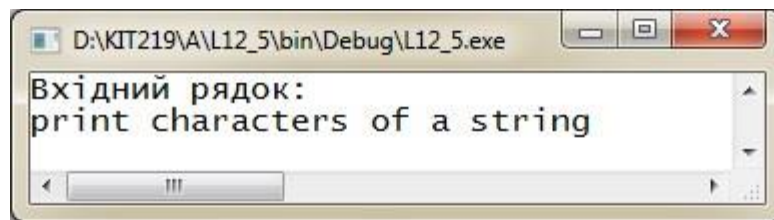


Рисунок 6.17 – Приклад передачі вказівника на константні дані

Оголошення читається справа наліво як «**sPtr** – це вказівник на символьну константу». Функція за допомогою інструкції **for** виводить символи рядка, поки не зустріне нульовий символ. Після виводу кожного символу вказівник **sPtr** пересувається на наступний символ у рядку.

Наступна програма демонструє функцію, що приймає вказівник (**xPtr**), що змінюється, на константні дані. Функція **f()** намагається змінити дані, на які вказує вказівник **xPtr**, що призводить до помилки компіляції. Фактичний текст повідомлення залежить від конкретного компілятора.

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

void f(const int *xPtr); // прототип

int main(void)
{
    int y;      // визначити змінну y

    f(&y);      // f намагається виконати заборонену зміну
    return 0;
}

// вказівник xPtr неможна використовувати для зміни
// значення змінної, на яку він вказує
```

```c
void f(const int *xPtr)
{
      // помилка: неможливо змінити
      // константний об'єкт
      *xPtr = 199;
}
```

Результат роботи компілятора наведено на рис. 6.18.

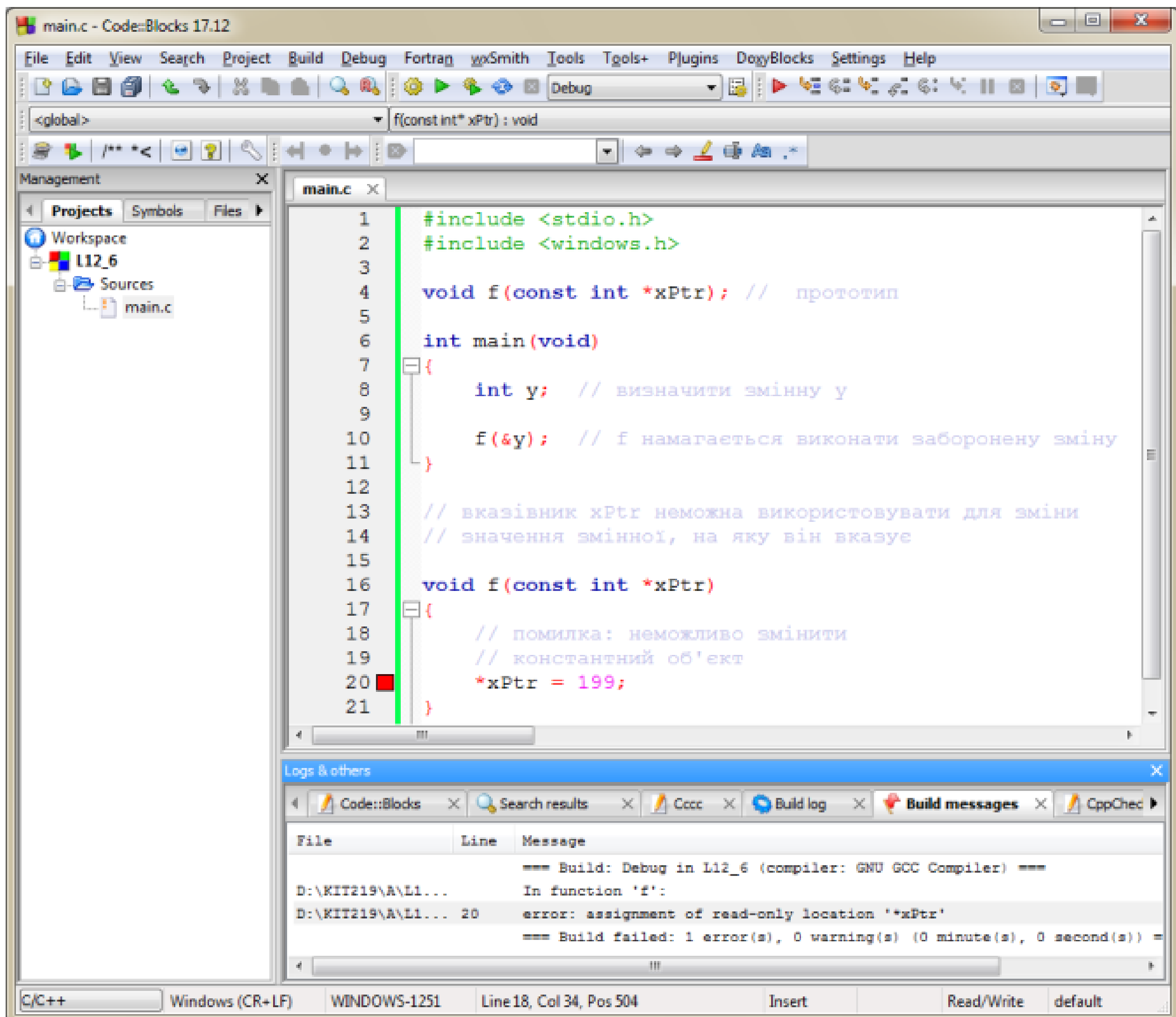


Рисунок 6.18 – Повідомлення компілятора про неможливість зміни значення **`*xPtr`**

Як відомо, масиви належать до складених типів даних, які зберігають під одним іменем елементи даних одного типа, які пов’язані між собою. Пізніше ми ознайомимося з ще одним представником складених типів даних – структурами (іноді в інших мовах програмування їх називають записами).

Структура може зберігати під одним іменем взаємопов’язані елементи

даних різних типів (наприклад, інформацію про кожного співробітника компанії). Коли в якості аргументу функції передається масив, цей масив автоматично передається за посиланням. Але структури завжди передаються за значенням – функція отримує копію структури. На копіювання елементів структури витрачається процесорний час. Тому, коли треба передати структуру даних, для економії часу можна передати функції вказівник на константні дані і отримати швидкість передачі за посиланням і захищеність передачі за значенням. При передачі вказівника на структуру копіюється тільки адреса структури. В системі з 4-х байтними адресами в даному випадку достатньо скопіювати усього 4 байти замість усієї структури, розмір якої може бути незрівнянно більше.

Структура може зберігати під одним іменем взаємопов'язані елементи даних різних типів (наприклад, інформацію про кожного співробітника компанії). Коли в якості аргументу функції передається масив, цей масив автоматично передається за посиланням. Але структури завжди передаються за значенням – функція отримує копію структури. На копіювання елементів структури витрачається процесорний час. Тому, коли треба передати структуру даних, для економії часу можна передати функції вказівник на константні дані і отримати швидкість передачі за посиланням і захищеність передачі за значенням. При передачі вказівника на структуру копіюється тільки адреса структури. В системі з 4-х байтними адресами в даному випадку достатньо скопіювати усього 4 байти замість усієї структури, розмір якої може бути незрівнянно більше.

Передача великих об'єктів, таких як структури, з використанням вказівників на константні дані дозволяє отримати переваги продуктивності передачі за посиланням і захищеність передачі за значенням.

При нестачі пам'яті та високих вимогах до швидкодії використовуйте вказівники. Якщо пам'яті достатньо і не треба забезпечити максимальну швидкодію, передавайте дані за значенням, втілюючи у життя принцип найменших привілеїв. Майте на увазі, що деякі системи підтримують

кваліфікатор типу **const** недостатньо добре, тому передача за значенням є кращим способом запобігти зміну даних.

Спроба змінити константний вказівник на дані, що змінюються. Константний вказівник на дані, що змінюються, завжди посилається на одну й ту саму адресу в пам'яті, і дані, що знаходяться там, можна змінити за допомогою цього вказівника. Прикладом такого вказівника може служити ім'я масиву. Ім'я масиву постійно вказує на його початок. Усі дані в масиві доступні та можуть змінюватися з використанням імені цього масиву та індексів. Константний вказівник на дані, що змінюються, можна використовувати у функціях для прийому аргументів-масивів, що забезпечують доступ до елементів масивів за допомогою індексів. Вказівники, які оголошені з кваліфікатором **const**, повинні ініціалізуватися в інструкції визначення (якщо вказівник є параметром функції, він ініціалізується вказівником, що передається до виклику функції).

Розглянемо програму, яка реалізує намагання змінити константний вказівник **ptr** типу **int * const**. Це визначення можна прочитати справа наліво як «**ptr** – це константний вказівник на ціле число». Текст програми має такий вигляд:

```
#include <stdio.h>

int main(void)
{
    int x;
    int y;
    // ptr - константний вказівник на ціле число,
    // яке можна змінити за допомогою ptr, але
    // ptr завжди вказує на адресу, яка не змінюється

    int * const ptr = &x;
    *ptr = 7; // допустимо: *ptr не є константою
    ptr = &y; // помилка: ptr - константа;
              // неможна присвоїти нову адресу

    return 0;
}
```

Результат роботи компілятора наведено на рис. 6.19.

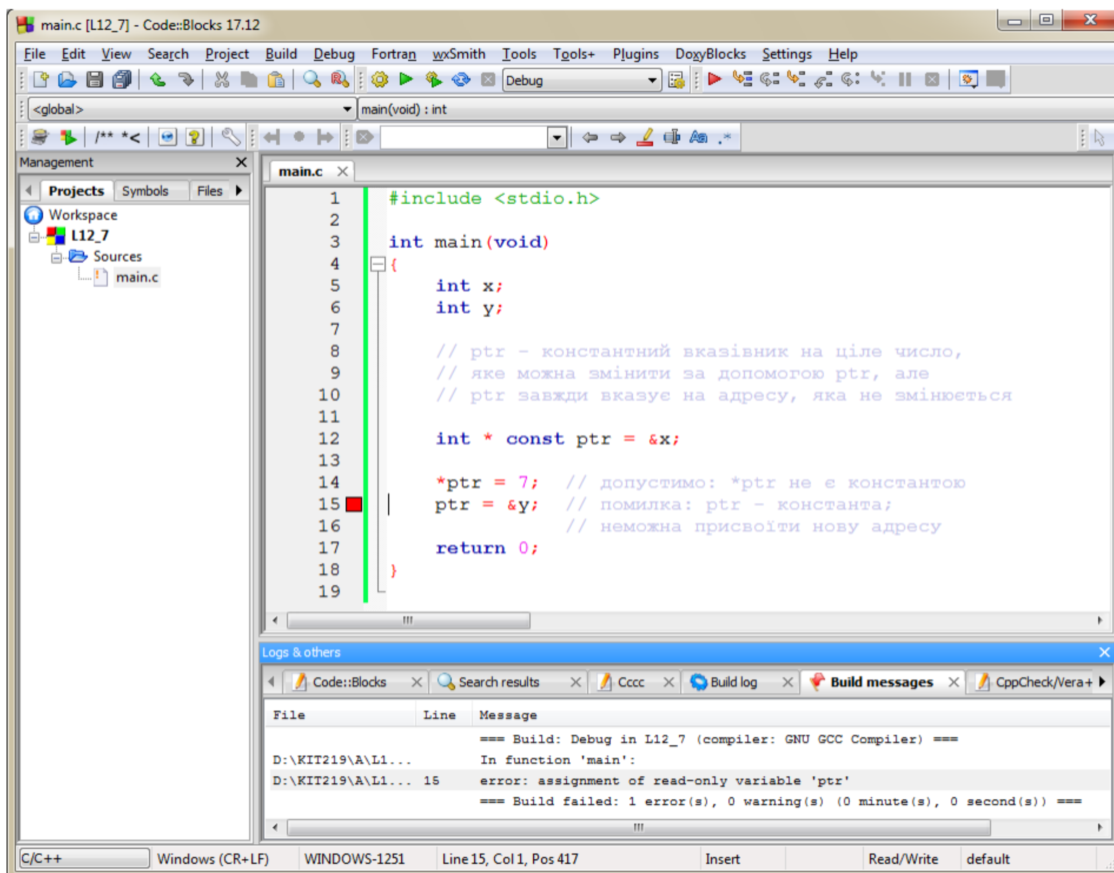


Рисунок 6.19 – Повідомлення компілятора про неможливість зміни адреси

Вказівник ініціалізується адресою цілочислової змінної **x**. Програма намагається присвоїти вказівнику **ptr** адресу змінної **y**, що призводить до помилки компіляції.

Намагання змінити константний вказівник на константні дані. Менш за все привілеїв доступу до даних дає константний вказівник на константні дані. Такий вказівник завжди посилається на одну й ту ж саму адресу в пам’яті та не дозволяє за його допомогою змінювати дані, які зберігаються за цією адресою. Саме так повинні передаватися масиви функціям, які тільки переглядають їх вміст з використанням індексів.

Розглянемо програму, яка визначає вказівник **ptr** типу **const int * const**. Це визначення читається справа наліво як «**ptr** – константний вказівник на цілочислову константу». Текст програми має такий вигляд:

```
#include <stdio.h>
int main(void)
```

```
{
    int x = 5;
    int y;

    // ptr - константний вказівник на цілочислову константу
    // ptr завжди вказує на адресу, яка не змінюється
    // ціле число за цією адресою не може бути змінено

    const int *const ptr = &x;    // ініціалізація виконується

    printf("%d\n", *ptr);
    *ptr = 7;   // помилка: *ptr - константа;
                // неможна присвоїти нове значення
    ptr = &y;   // помилка: ptr - константа;
                // неможна присвоїти нову адресу
}
```

Результат роботи компілятора наведено на рис. 6.20.

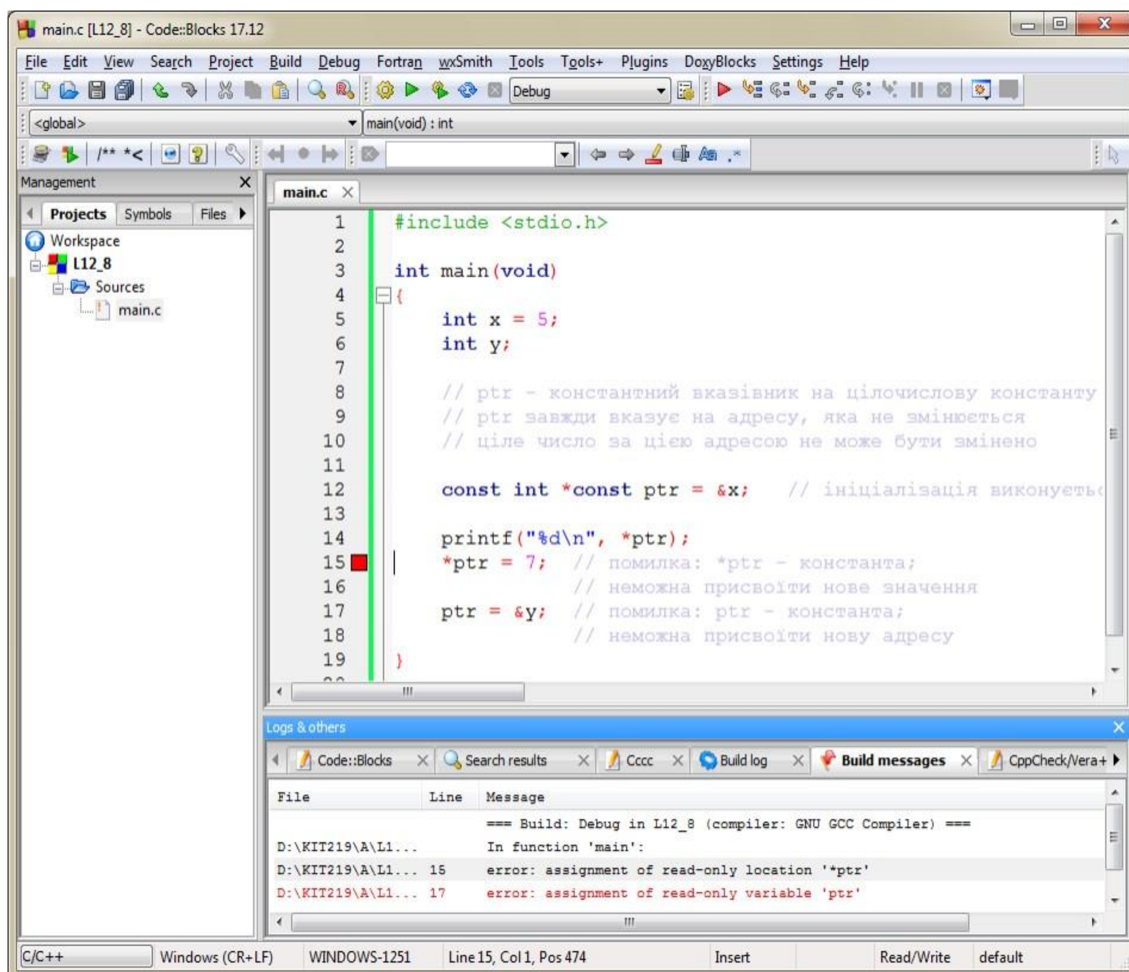


Рисунок.6.20 – Повідомлення компілятора про неможливість змін

На рис. 6.20 показані повідомлення про помилки, які видані компілятором для рядків у програмі, де відбувається спроба змінити дані, на

які вказує **ptr** (рядок 15), і змінити адресу, яка зберігається у змінній-вказівнику (рядок 17).

Бульбашкове сортування з передачею аргументів за посиланням. Розглянемо програму бульбашкового сортування, задіявши в ній дві функції **bubbleSort()** і **swap()**. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#define SIZE 10

void bubbleSort(int * const array, size_t size); // прототип

int main(void)
{
    // ініціалізація масиву a[]
    int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
    size_t i;       // лічильник

    SetConsoleOutputCP(1251);
    puts("Порядок елементів вхідного масиву");
    // обійти елементи масиву a[]
    for(i = 0; i < SIZE; i++)
       printf( "%4d", a[i]);
  bubbleSort(a, SIZE); // відсортувати масив
  puts( "\n\nПорядок елементів вихідного масиву");
  // обійти елементи масиву a[]
  for(i = 0; i < SIZE; i++)
      printf("%4d", a[i]);
  puts("");
  return 0;
}

// сортує масив чисел з використанням алгоритму
// бульбашкового сортування

void bubbleSort(int * const array, size_t size)
{
  void swap(int *element1Ptr, int *element2Ptr); // прототип
  unsigned int pass;   // лічильник проходів
  size_t j;            // лічильник порівнянь

  // цикл, який керується лічильником проходів
  for(pass = 0; pass < size - 1; pass++)
  {
      // цикл, який керується лічильником порівнянь
      // в кожному проході
      for(j = 0; j < size - 1; j++)
      {
          // обміняти місцями сусідні елементи,
```

```
                // якщо це необхідно
                if(array[j] > array[j+1])
                {
                    swap(&array[j], &array[j+1]);
                }
            }
        }
    }

    // міняє місцями значення в пам'яті, які адресуються
    // вказівниками element1Ptr и element2Ptr

void swap(int *element1Ptr, int *element2Ptr)
{
    int hold = *element1Ptr;

    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
}
```

Результат виконання програми наведено на рис. 6.21.

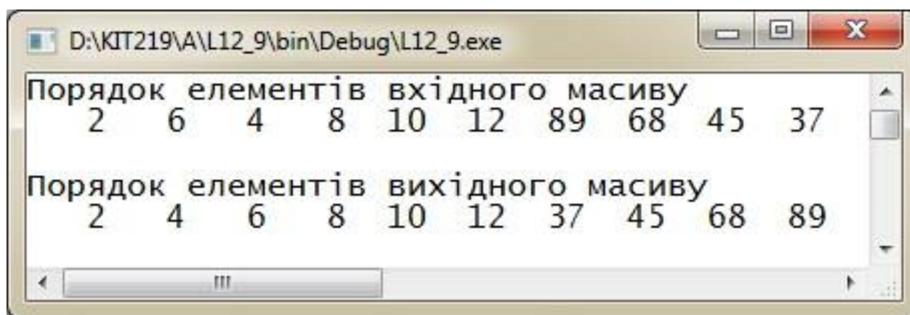


Рисунок 6.21 – Використання функцій для бульбашкового сортування

Функція **bubbleSort()** сортує масив. Для обміну місцями елементів масивів **array[j]** і **array[j+1]** вона викликає функцію **swap()**. В мові **C** функції приховують інформацію одна від одної, тому **swap()** не має доступу до окремих елементів масиву, що обробляється функцією **bubbleSort()**. Оскільки функції **bubbleSort()** треба, щоб функція **swap()** мала доступ до елементів масиву, які міняються місцями, **bubbleSort()** передає обидва ці елементи за посиланням, тобто явно передає адреси обох елементів. Незважаючи на те, що при передачі масиву цілком він автоматично передається за посиланням, окремі елементи масиву та скаляри за замовчуванням передаються за значенням. Саме тому в

bubbleSort() застосовується оператор взяття адреси (**&**) для кожного елемента масиву в виклику функції **swap()**:

```
swap(&array[j], &array[j+1]);
```

Функція **swap()** приймає аргумент **&array[j]** у вигляді змінної-вказівника **element1Ptr**. Навіть не дивлячись на те що ім'я **array[j]** невідоме функції **swap()**, вона все ж може використовувати ім'я ***element1Ptr** як синонім для **array[j]** – звертаючись до ***element1Ptr** функція **swap()** фактично звертається до **array[j]** у **bubbleSort()**. Аналогічно, коли **swap()** звертається до ***element2Ptr**, вона фактично звертається до **array[j+1]** у **bubbleSort()**. Навіть при тому, що у функції **swap()** неможна записати:

```
int hold = array[j];
array[j] = array[j+1]
array[j+1]; = hold;
```

такий саме ефект досягається рядками:

```
int hold = *element1Ptr;

*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

Слід також відзначити деякі особливості функції **bubbleSort()**. Заголовок функції оголошує масив як **int * const array** замість **int array[]**, показуючи тим самим, що **bubbleSort()** приймає одновимірний масив. Параметр **size** оголошений з кваліфікатором **const**, відповідно до принципу найменших привілеїв. Незважаючи на те що в параметрі **size** передається копія значення змінної в **main()** і зміна копії не вплине на фактичне значення змінної в **main()**, все ж таки, функції **bubbleSort()** не треба змінювати **size** для рішення її задачі. Розмір масиву залишається постійним в ході виконання **bubbleSort()**. Тому параметр **size** оголошений з

кваліфікатором `const`, щоб гарантувати його незмінність.

Прототип функції `swap()` включений до тіла функції `bubbleSort()`. Зроблено це тому, що `bubbleSort()` є єдиною функцією, яка викликає `swap()`. Розміщення прототипу в `bubbleSort()` обмежує можливість виклику `swap()` рамками `bubbleSort()`. якщо інші функції в програмі, що не мають доступу до прототипу `swap()`, намагаються викликати її, компілятор згенерує невідомий прототип автоматично. Зазвичай такий автоматично згенерований прототип не збігається з заголовком функції (що призводить до помилки або до попередження на етапі компіляції), тому що компілятор передбачає, що функція повинна приймати та повертати значення типу `int`.

Розміщення прототипу у визначенні іншої функції сприяє просуванню принципу найменших привілеїв, обмежуючи можливість виклику функції тільки там, де є присутнім її прототип.

Функція `bubbleSort()` приймає розмір масиву у вигляді параметра. Функція повинна знати розмір масиву, щоб відсортувати його. Коли функції передається масив, вона отримує адресу першого елемента в пам'яті. Звичайно, за адресою неможливо визначити кількість елементів в масиві. Тому необхідно також передати розмір масиву. Часто на практиці використовується інший прийом, коли функції передається вказівник на початок масиву та вказівник на першу комірку пам'яті безпосередньо за останнім елементом масиву, різниця двох вказівників визначає довжину масиву, а отриманий код виглядає простіше.

У попередній програмі розмір масиву явно передається функції `bubbleSort()`. Такий підхід має ще дві переваги – *можливість повторного використання коду* і *надійність програмної архітектури*. Визначаючи функцію, яка приймає розмір масиву в аргументі, забезпечують можливість її використання в будь-яких програмах, які оперують одновимірними масивами довільного розміру.

Передаючи функції масив, передавайте також його розмір. Це допоможе спростити використання даної функції в інших програмах.

Можна зберігати розмір масиву, що сортується, в глобальній змінній, яка доступна з будь-якої точки в програмі. Це трохи б підвищило ефективність за рахунок відсутності операції копіювання розміру та передачі його функції. Однак в інших програмах, де необхідна можливість сортування масивів цілих чисел, подібна глобальна змінна може бути відсутньою, і тому вони не зможуть використовувати дану функцію.

Використання глобальних змінних часто призводить до порушення принципу найменших привілеїв і зниженню надійності програм. Глобальні змінні слід використовувати тільки для представлення ресурсів, які розділяються по-справжньому, таких як поточний час.

Розмір масиву можна було б також «зашити» безпосередньо у функції. Однак подібний прийом зробив би неможливим застосування функції до масивів іншого розміру та суттєво обмежив би область її використання. Таку функцію можна було б використовувати лише у програмах, які оброблюють одновимірні масиви того ж самого розміру.

6.2.5. Вирази з вказівниками та арифметика вказівників

Вказівники є допустимими операндами в арифметичних виразах, виразах присвоювання та порівняння. Однак далеко не всі оператори, що зазвичай застосовуються у виразах, можуть використовуватися у виразах з вказівниками.

Для виконання операцій з вказівниками допускається використовувати лише обмежену кількість арифметичних операторів. Вказівники можуть збільшуватися за допомогою оператора інкремента (`++`) або зменшуватися за допомогою оператора декремента (`--`). До вказівників можна додавати цілі числа (`+` і `+=`). З вказівників можна віднімати цілі числа (`-` і `-=`), і один вказівник може відніматися від іншого. Остання операція має сенс, тільки якщо обидва вказівника вказують на елементи одного й того ж самого масиву.

Припустимо, що визначено масив `int v[5]`, а його перший елемент зберігається у пам’яті за адресою `3000`. Припустимо, що

вказівник **vPtr** ініціалізований адресою першого елемента **v[0]**, тобто **vPtr** має значення **3000**. Ця ситуація на комп'ютері з 4-х байтними цілими числами, зображена на рис. 6.22.

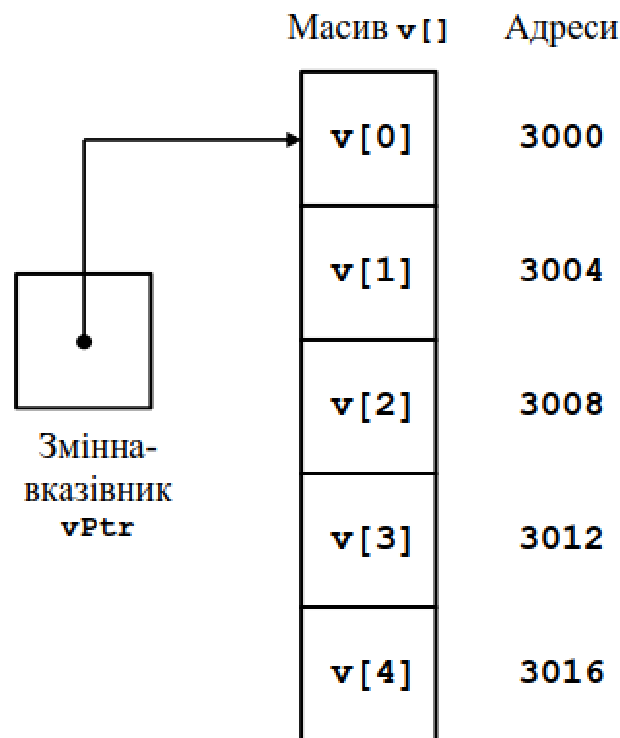


Рисунок 6.22 – Масив **v** і змінна-вказівник **vPtr**, що посилається на **v**

Змінна **vPtr** може бути ініціалізована вказівником на масив **v** однією з наступних інструкцій:

```
vPtr = v;
vPtr =
&v[0];
```

Оскільки результати арифметичної операції з вказівником залежать від розміру об'єкта, на який посилається вказівник, арифметика вказівників значною мірою залежить від апаратної та програмної архітектури комп'ютера.

У звичайній арифметиці вираз **3000 + 2** дає в результаті **3002**. Але в арифметиці вказівників це не завжди так. Коли ціле число додається до вказівника або віднімається від нього, вказівник не просто збільшується або зменшується на це число, а на дане число, помножене на розмір об'єкта, на

який посилається вказівник. Розмір об'єкта залежить від його типа. Наприклад, інструкція

```
vPtr += 2;
```

присвоїть змінній **vPtr** адресу **3008** (**3000 + 2*4**), якщо виходити з припущення, що значення типу **int** займає **4** байти. Тепер **vPtr** буде вказувати на елемент масиву **v[2]** (рис. 6.23).

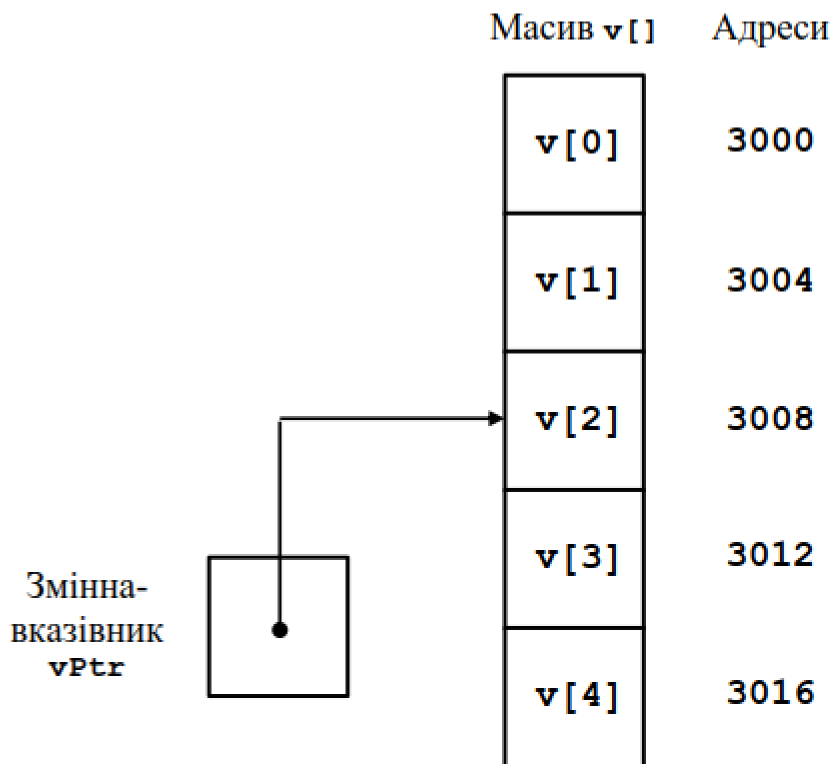


Рисунок 6.23 – Вказівник **vPtr** після арифметичної операції `vPtr += 2`

В системі, де значення типу **int** займає **2** байти, попередня інструкція присвоїла б змінній **vPtr** адресу **3004** (**3000 + 2*2**). Якщо б масив **v** зберігав елементи іншого типу, попередня інструкція збільшила б адресу в змінній **vPtr** на кількість байтів, яку займає цей тип, помножену на два. При виконанні арифметичних операцій з вказівниками на масиви символів результати будуть відповідати звичайним арифметичним операціям, тому що один символ займає **1** байт.

Якщо в будь-який момент часу вказівник **vPtr** був збільшений до значення **3016**, що відповідає адресі елемента **v[4]**, тоді інструкція

```
vPtr -= 4;
```

зменшить його до значення **3000** – адреса початку масиву. Для збільшення або зменшення вказівника на одиницю можна використовувати оператори інкремента (**++**) та декремента (**--**). Будь-яка з наступних інструкцій

```
++vPtr;
vPtr++;
```

збільшить вказівник, пересунувши його на наступний елемент масиву. Будь-яка з наступних інструкцій

```
--vPtr;
vPtr--;
```

зменшить вказівник, пересунувши його на попередній елемент масиву.

Змінні-вказівники можуть відніматися один з одного. Наприклад, якщо припустити, що **vPtr** зберігає адресу **3000**, а **v2Ptr** – адресу **3008**, тоді інструкція

```
x = v2Ptr - vPtr;
```

присвоїть змінній **x** кількість елементів масиву від **vPtr** до **v2Ptr**, в даному випадку **2** (не **8**). Арифметика вказівників не визначена, якщо вказівники не посилаються на масив. Ми не можемо припустити, що дві змінні одного типу зберігаються в пам'яті по сусідству, якщо тільки вони не є сусідніми елементами масиву.

Зазвичай є помилкою:

1) застосовувати арифметику вказівників до вказівника, який посилається не на елемент масиву;

2) віднімати або порівнювати два вказівника, які посилаються не на елементи одного масиву;

3) виходити за межі масиву, застосовуючи арифметику вказівників. Значення змінної-вказівника може бути присвоєно іншій змінній-вказівнику,якщо вони є змінними одного й того ж самого типу. Виключенням з цього правила є вказівник на значення типу **void** (тообто **void ***), який вважається універсальним вказівником, що здатний представляти вказівник будь-якого типу. Вказівнику типу **void *** можна присвоїти вказівник

будь-якого типу, і вказівник типу **void *** можна присвоїти вказівнику будь-якого типу. В жодному з цих випадків не треба виконувати операцію приведення.

Вказівник типу **void *** не може бути **розіменований**. Уявіть собі таку ситуацію: компілятор знає, що вказівник типу **int *** посилається на **4**-х байтний об'єкт в системі, де значення типу **int** займає **4** байти, а вказівник типу **void *** просто містить адресу значення невідомого типу – точна кількість байтів, яку займає об'єкт, на який посилається вказівник, компілятору невідома. Однак, щоб розіменувати вказівник, компілятор повинен знати, скільки байтів займає об'єкт, що адресується вказівником.

Спроба присвоїти значення вказівника одного типа вказівнику іншого типу, якщо жоден з них не є вказівником типу **void *** призведе до синтаксичної помилки.

Спроба розіменувати вказівник типу **void *** призведе до синтаксичної помилки.

Вказівники можуть порівнюватися за допомогою операторів порівняння і операторів відношення, але таке порівняння не має сенсу, якщо вказівники не посилаються на елементи одного масиву. При порівнянні вказівників порівнюються адреси в пам'яті. Порівняння двох вказівників, що посилаються на елементи одного масиву, може допомогти визначити, наприклад, який з них посилається на елемент з більшим індексом. Частіше за все операція порівняння з вказівниками застосовується, щоб визначити, чи не містить вказівник значення **NULL**.

6.2.6. Зв'язок між вказівниками та масивами

Масиви та вказівники в мові **C** мають дуже тісний зв'язок і часто можуть використовуватися заміняючи один одного. Ім'я масиву можна розглядати як константний вказівник. Вказівники можна використовувати для виконання будь-яких операцій, що застосовують індекси.

Припустимо, що в програмі визначені масив цілих чисел **b[5]** і вказівник

на ціле число **bPtr**. Оскільки ім’я масиву (без індексу) є вказівником на перший елемент, можна присвоїти змінній **bPtr** адресу першого елемента в масиві **b** інструкцією

```
bPtr = b;
```

Ця інструкція еквівалентна операції взяття адреси першого елемента масиву:

```
bPtr = &b[0];
```

За бажанням доступ до елемента **b[3]** можна отримати за допомогою виразу

```
*(bPtr+3)
```

Число **3** в даному виразі – це **зміщення відносно вказівника**. Якщо вказівник посилається на перший елемент масиву, зміщення визначає адресу елемента того ж самого масиву, що повертається виразом, і у цьому випадку значення зміщення еквівалентно індексу елемента в масиві. Таку форму запису часто називають **вказівник/зміщення**. Круглі скобки необхідні лише тому, що оператор ***** має більш високий пріоритет, ніж оператор **+**. Без круглих скобок вираз, який наведено вище, додав би число **3** до значення виразу ***bPtr** (тобто до до значення елемента **b[0]**, якщо виходити з припущення, що змінна **bPtr** вказуж на перший елемент масиву). Вираз взяття адреси елемента масиву

```
&b[3]
```

можна також записати як

```
bPtr + 3
```

Ім’я самого масиву також можна інтерпретувати як вказівник і застосовувати до нього арифметику вказівників. Наприклад, вираз

```
*(b + 3)
```

також поверне значення елемента **b[3]**. Взагалі, будь-які вирази за участі індексів елементів масивів можна записати за допомогою вказівників і зміщень.

В даному випадку форма запису вказівник/зміщення була застосована до імені масиву. Попередня інструкція не змінює значення імені масиву; **b** як і раніше вказує на перший елемент.

Вказівники можна використовувати з індексами, подібно до імен масивів. Якщо припустити, що **bPtr** зберігає адресу **b**, вираз

```
bPtr[1]
```

буде посилатися на елемент **b[1]**. Цю форму запису часто називають вказівник/індекс.

Ім’я масиву фактично є константним вказівником. Воно завжди вказує на початок масиву. Тобто вираз

```
b += 3
```

є недопустимим, тому що він намагається змінити значення імені масиву з застосуванням арифметики вказівників.

Намагання змінити значення імені масиву за допомогою арифметики вказівників призводить до помилки компіляції.

В наступній програмі демонструються чотири способи звернення до елементів масиву, які розглядалися вище, – з застосуванням індексу до імені масиву, з використанням форми запису вказівник/зміщення з іменем масиву, з застосуванням індексу до вказівника та з використанням форми запису вказівник/зміщення з вказівником – для виводу чотирьох елементів масиву **b**.

```
#include <stdio.h>
#include <windows.h>
#define ARRAY_SIZE
int main(void)
{
   // створити та ініціалізувати масив b
   int b[] = { 10,  20,  30, 40 };
   // створити вказівник і присвоїти йому адресу масиву b
    int *bPtr = b;
    size_t i;            // лічильник
    size_t offset;       // лічильник

    SetConsoleOutputCP(1251);
```

```
    // вивести масив b з використанням індексів
    puts("Масив b:\n\n"
         "Запис за допомогою індексів");
    // цикл по елементам масиву b
    for(i = 0; i < ARRAY_SIZE; ++i)
        printf("b[%u] = %d;\n", i, b[i]);
    // вивести масив b з використанням форми запису
    // вказівник/зміщення
    puts("\nЗапис вказівник/зміщення, де вказівником"
         " є ім\'я масиву");
    // цикл по елементам масиву b
    for(offset = 0; offset < ARRAY_SIZE; offset++)
       printf("*(b + %u) = %d;\n", offset, *(b + offset));
    // вивести масив b з використанням bPtr і індексів
    puts("\nЗапис вказівник/індекс");
    // цикл по елементам масиву b
    for(i = 0; i < ARRAY_SIZE; ++i)
        printf("bPtr[%u] = %u;\n", i, bPtr[i]);
    // вивести масив з використанням bPtr і форми
    // запису вказівник/зміщення
    puts("\nЗапис вказівник/зміщення");
    // цикл по елементам масиву b
    for(offset = 0; offset < ARRAY_SIZE; ++offset)
        printf("*(bPtr + %u) = %d;\n",
               offset, *(bPtr + offset));
    return 0;
}
```

Результат виконання програми наведено на рис. 6.24.

```
D:\KIT219\A\L12_10\bin\Debug\L12_10.exe
Масив b:

Запис за допомогою індексів
b[0] = 10;
b[1] = 20;
b[2] = 30;
b[3] = 40;

Запис вказівник/зміщення, де вказівником є ім'я масиву
*(b + 0) = 10;
*(b + 1) = 20;
*(b + 2) = 30;
*(b + 3) = 40;

Запис вказівник/індекс
bPtr[0] = 10;
bPtr[1] = 20;
bPtr[2] = 30;
bPtr[3] = 40;

Запис вказівник/зміщення
*(bPtr + 0) = 10;
*(bPtr + 1) = 20;
*(bPtr + 2) = 30;
*(bPtr + 3) = 40;
```

Рисунок.6.24 – Демонстрація чотирьох способів звернення до елементів масиву

Копіювання рядків із застосуванням масивів і вказівників. Для подальшої демонстрації використання масивів і вказівників розглянемо програму, яка використовує дві функції копіювання рядків – **copy1()** і **copy2()**. Обидві функції копіюють рядок до масиву символів. Прототипи функцій **copy1()** і **copy2()** виглядають ідентичними. Вони вирішують одну й ту ж саму задачу, але реалізовані по-різному.

Текст програми для копіювання рядка з використанням форми запису звернення до масиву та вказівника має такий вигляд:

Текст програми для копіювання рядка з використанням форми запису звернення до масиву та вказівника має такий вигляд:

```
#include <stdio.h>
#define SIZE 10
void copy1(char * const s1, const char* const s2); // прототип
void copy2(char *s1, const char *s2);              // прототип

int main(void)
{
    char string1[SIZE];           // створити масив string1
    char *string2 = "Hello";      // створити вказівник на рядок
    char string3[SIZE];           // створити масив string3
    char string4[] = "Good Bye"; // створити вказівник на рядок
    copy1(string1, string2);
    printf("string1 = %s\n", string1);
    copy2(string3, string4);
    printf("string3 = %s\n", string3);
    return 0;
}

// копіює s2 в s1 з використанням форми
// запису звернення до масиву

void copy1(char * const s1, const char * const s2)
{
    size_t i;      // лічильник
    // цикл по символам в рядках
    for(i = 0; (s1[i] = s2[i]) != '\0'; ++i)
    {
        ; // порожнє тіло циклу
    }
}
// копіює s2 в s1 з використанням форми
// запису звернення до вказівника

void copy2(char *s1, const char *s2)
{
```

```
    // цикл по символам в рядках
    for( ; (*s1 = *s2) != '\0'; ++s1, ++s2)
    {
        ; // порожнє тіло циклу
    }
}
```

Результат виконання програми наведено на рис. 6.25.

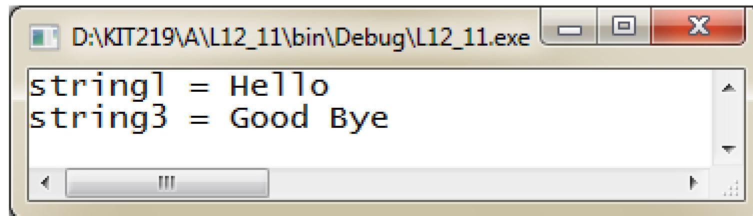


Рисунок 6.25 – Копіювання рядка з використанням форми запису звернення до масиву та вказівника

Для копіювання рядка **s2** до масиву символів **s1** функція **copy1()** використовує форму звернення до елементів масиву. Для переліку індексів функція визначає змінну-лічильник **i**. Власне операція копіювання виконується заголовком інструкції **for** – тіло циклу складає порожня інструкція. В заголовку цієї інструкції **for** змінна **i** ініціалізується нулем і збільшується на **1** в кожній ітерації циклу. Вираз **s1[i] = s2[i]** копіює один символ з **s2** в **s1**. Коли в **s2** зустрічається нульовий символ, він копіюється в **s1** і результат операції присвоювання стає значенням лівого операнда оператора порівняння. В результаті умова продовження циклу стає хибною і копіювання припиняється.

Для копіювання рядка **s2** до масиву символів **s1** функція **copy2()** використовує форму звернення до вказівника та арифметику вказівників. Знов таки, копіювання виконується заголовком інструкції **for**. На цей раз заголовок інструкції не виконує ініціалізацію будь-яких змінних. Як і в функції **copy1()**, копіювання виконується виразом (***s1 = *s2**). Цей вираз розіменовує вказівник **s2** і отриманий символ присвоює розіменованому вказівнику ***s1**. Після присвоювання в умовному виразі вказівники переміщуються операцією інкремента на наступний символ: **s1** – в масиві та **s2** –

в рядку відповідно. Коли в рядку **s2** зустрічається нульовий символ, він присвоюється розіменованому вказівнику **s1**, і цикл завершується.

В першому аргументі обом функціям – **copy1()** і **copy2()** – повинен передаватися масив достатнього розміру, щоб вмістити рядок, передану в другому аргументі. В іншому випадку під час намагання запису за межами масиву може виникнути помилка. Зверніть також увагу, що другий параметр в обох функціях оголошений як **const char *** (константний рядок). Обидві функції копіюють другий аргумент у перший – вони читають символи з другого аргументу по одному, але не змінюють їх. Тобто другий параметр оголошений як вказівник на константні дані, виходячи з принципу найменших привілеїв – жодній з функцій не треба змінювати другий аргумент, тому жодній х них не надається така можливість.

Масиви вказівників. Масиви можуть зберігати не тільки звичайні значення, але й вказівники. Часто можливість створення масивів вказівників використовується для визначення масивів рядків.. Кожен елемент такого масиву є рядком, але в мові **C** рядок фактично є вказівником на перший символ цього рядка. Тобто кожен елемент масиву рядків в дійсності є вказівником. Погляньте на визначення масиву рядків **suit**, який може бути корисним для представлення колоди гральних карт:

```
const char *suit[4] = {"Hearts","Diamonds","Clubs","Spades"};
```

Фрагмент **suit[4]** визначення вказує, що масив містить **4** елемента. Фрагмент **char *** повідомляє, що кожен елемент масиву **suit** має тип «вказівник на символ». Кваліфікатор **const** говорить, що рядки, на які вказують елементи масиву, не можуть змінюватися за їх допомогою. Масив ініціалізований чотирма елементами: **"Hearts"**, **"Diamonds"**, **"Clubs"** і **"Spades"**. Кожен зберігається в пам’яті як рядок, що завершується нульовим символом, на один символ довше, ніж рядки, що розміщені в лапках. Рядки мають довжину **7**, **9**, **6** и **7** символів відповідно. На перший погляд, здається, що масив зберігає фактичні рядки, однак в дійсності його елементами є

вказівники (рис. 6.26). Кожен вказівник посилається на перший символ відповідного рядка. Тобто навіть при тому, що масив **suit** має фіксований розмір, він забезпечує доступ до рядків довільної довжини. Така гнучкість є одним з прикладів широких можливостей представлення структур даних в мові **C**.

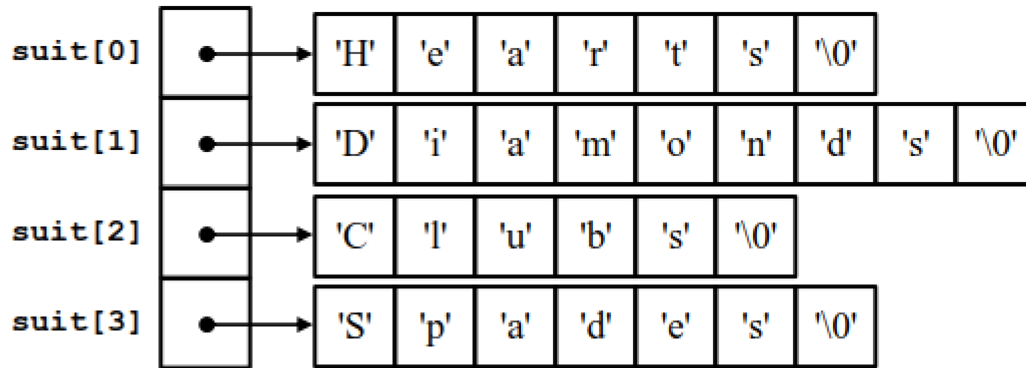


Рисунок 6.26 – Графическое представление массива **suit**

Ту же саму інформацію можна було б помістити в двовимірний масив, в якому кожний рядок представляє окрему масть в картах, а кожен стовпець – букву в назві масті. При такій організації структури даних прийшлося б визначити фіксовану кількість стовпців, достатню для зберігання самого довгого з можливих рядків. Подібний підхід може призводити до марному витрачанню пам'яті, коли треба зберігати велику кількість рядків, значна частина яких суттєво коротше самого довгого рядка.

6.2.7. Вказівники на функції

Вказівник на функцію зберігає адресу функції у пам'яті. Ім'я масиву в дійсності є вказівником на перший елемент цього масиву. Аналогічно ім'я функції в дійсності є адресою першої інструкції в тілі функції. Вказівники на функції можна передавати іншим функціям, повертаючись з функцій, зберігати в масивах і присвоювати іншим вказівникам на функції.

Для ілюстрації використання вказівників на функції в наступній програмі приводиться модифікована версія програми бульбашкового

сортування. Дана версія складається з функції **main()**, а також містить функції **bubble()**, **swap()**, **ascending()** і **descending()**. Окрім масиву цілих чисел і розміру масиву, функція **bubble()** приймає також вказівник на функцію – **ascending()** або **descending()**. Програма пропонує користувачеві обрати порядок сортування масиву за зростанням або за убуванням. Якщо користувач введе число **1**, до функції **bubble()** передається вказівник на функцію **ascending()**, в результаті чого масив сортується в порядку зростання значень елементів. Якщо користувач введе **2**, до функції **bubble()** передається вказівник на функцію **descending()**, і масив сортується в порядку убування. Якщо користувач введе **3** – програма завершиться. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#define SIZE 10

// прототипи
void bubble(int work[], size_t size, int (*compare)(int a, int b));
int ascending(int a, int b);
int descending(int a, int b);

int main(void)
{
    int order;              // 1 - за зростанням, 2 — за зменшенням
    size_t counter;         // лічильник

    SetConsoleOutputCP(1251);

    while(1)
    {
        // ініціалізувати неупорядкований масив a
        int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
        puts("============================");
        printf("%s", "1 - сортувати за зростанням\n"
                     "2 - сортувати за зменшенням\n"
                     "3 - завершити програму\n");
        puts("============================");
        scanf("%d", &order);
        if(order == 3) exit(0);

        puts("\nЕлементи вхідного масиву");
        // Вивести вхідний масив
        printf("==============================================\n");
        for(counter = 0; counter < SIZE; counter++)
```

```
            printf("%5d", a[counter]);
        printf("\n============================================\n");

        // сортувати масив за зростанням
        // передати функцію ascending() в аргументі,
        // щоб забезпечити сортування за зростанням
        switch(order)
        {
            case 1: bubble(a, SIZE, ascending); puts("\nЕлементи
                    масиву за зростанням");
                    break;
            case 2: bubble(a, SIZE, descending);
                    puts("\nЕлементи масиву за зменшенням");
                    break;
           default: break; }
        // вивести відсортований масив
        printf("==============================================\n");
        for(counter = 0; counter < SIZE; counter++)
            printf("%5d", a[counter]);
        printf("\n============================================\n");
        printf("\n\n");
    }
    return 0;
}

//=======================================================
// Універсальна функція сортування параметр
// compare - це вказівник на функцію порівняння,
// яка визначає порядок сортування
//=======================================================
void bubble(int work[], size_t size, int (*compare)(int a, int b))
{
    unsigned int pass;  // лічильник проходів
    size_t count;       // лічильник порівнянь
    void swap(int *element1Ptr, int *element2ptr); // прототип
    // цикл, який керує кількістю проходів
    for(pass = 1; pass < size; pass++)
    {
        // цикл, який керує кількістю порівнянь в одному проході
        for(count = 0; count < size - 1; count++)
        {
            // якщо сусідні елементи стоять не в тому порядку,
            // поміняти їх місцями
            if((*compare)(work[count], work[count+1]))
            {
                swap(&work[count], &work[count+1]);
            }
        }
    }
}

//=======================================================
// Міняє місцями значення елементів, на які вказує
```

```
// element1Ptr i element2Ptr
//=========================================================
void swap(int *element1Ptr, int *element2Ptr)
{
    int hold;   // тимчасова змінна

    hold = *element1Ptr;
    *element1Ptr = *element2Ptr;
    *element2Ptr = hold;
}

//=========================================================
// Сортування за зростанням
//=========================================================
int ascending(int a, int b)
{
    return b < a;    // треба поміняти місцями,
                     // якщо b < a
}

int descending(int a, int b)
{
    return b > a;   // треба поміняти місцями,
                    // якщо b > a

}
```

Результат виконання програми наведено на рис. 6.27

В заголовку функції **bubble()** оголошений такий параметр:

```
int (*compare)(int a, int b)
```

Він повідомляє, що параметр (**compare**) є вказівником на функцію, яка має два цілочислових параметри та повертає цілочисловий результат. Круглі дужки навколо ***compare** необхідні, щоб згрупувати оператор ***** з іменем **compare** й тим самим показати, що **compare** є вказівником. якщо круглі дужки опустити, оголошення набуде наступного виду:

```
int *compare(int a, int b)
```

що рівносильно оголошенню функції, що приймає два цілих числа і повертає вказівник на ціле число.

```
D:\KIT219\A\Lab6_6\bin\Debug\Lab6_6.exe
============================
1 - сортувати за зростанням
2 - сортувати за зменшенням
3 - завершити програму
============================
1

Елементи вхідного масиву
===================================================
    2    6    4    8   10   12   89   68   45   37
===================================================

Елементи масиву за зростанням
===================================================
    2    4    6    8   10   12   37   45   68   89
===================================================


============================
1 - сортувати за зростанням
2 - сортувати за зменшенням
3 - завершити програму
============================
2

Елементи вхідного масиву
===================================================
    2    6    4    8   10   12   89   68   45   37
===================================================

Елементи масиву за зменшенням
===================================================
   89   68   45   37   12   10    8    6    4    2
===================================================


============================
1 - сортувати за зростанням
2 - сортувати за зменшенням
3 - завершити програму
============================
3
```

Рисунок 6.27 – Модифікована програми бульбашкового сортування

В прототипі функції **bubble()** визначення третього параметра має вигляд

```
int (*)(int, int);
```

без імені-вказівника на функцію і без імен параметрів.

Функція, яка передається функції **bubble()**, викликається в інструкції **if**:

```
if((*compare)(work[count], work[count+1]))
```

Так само як і для доступу до значення змінної необхідно розіменовувати вказівник, для виклику функції необхідно розіменувати вказівник на функцію.

Виклик функції можна провести і без розіменування вказівника, як, наприклад, в інструкції:

```
if(compare(work[count], work[count+1]))
```

де вказівник використовується безпосередньо як ім'я функції.

Краще використовувати перший спосіб виклику функцій за вказівниками, тому що він наочно демонструє, що **compare** – це вказівник на функцію, який розіменовується у виклик функції.

Другий спосіб робить ім'я вказівника **compare** таким, що не відрізняється від імені фактичної функції, що може спричинити незручність у тих, хто буде читати ваш код і, захоче подивитися визначення функції **compare**, і виявиться, що вона ніде не визначена.

Використання вказівників на функції для створення програм, які керуються системою меню. Вказівники на функції часто використовуються в текстових програмах, що керуються системою меню. користувачу пропонується обрати пункт меню (наприклад, від 1 до 5) шляхом вводу номера елемента меню. Кожен пункт обслуговується окремою функцією. Вказівники на ці функції зберігаються в масиві. Вибір користувача інтерпретується як індекс елемента в масиві, що використовується для виклику функції.

Розглянемо узагальнений приклад механізму визначення та використання масиву вказівників на функції. В наступній програмі визначаються три функції – **function1()**, **function2()** і **function3()**, кожна з яких приймає ціле число і нічого не повертає. Вказівники на ці функції зберігаються в масиві **f**. Текст програми має такий вигляд:

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

```
// прототипи
void function1(int a);
void function2(int b);
void function3(int c);

int main(void)
{
    // ініціалізувати масив з 3 вказівниками на функції,
    // кожна з яких приймає ціле число і нічого не повертає
    void (*f[3])(int) = { function1, function2, function3 };
    size_t choice; // змінна для зберігання вибору користувача

    SetConsoleOutputCP(1251);

    printf("%s", "Введіть число між 0 і 2, 3 - для виходу: " );
    scanf( "%u",&choice);

    // обробити вибір користувача
    while(choice >= 0 && choice < 3)
    {
        // викликати функцію за вказівником в елементі масиву f
        // з індексом choice і передати їй значення choice
        (*f[choice])(choice);
        printf("%s", "Введіть число між 0 і 2, 3 - для виходу: " );
        scanf("%u", &choice);
    }
    puts("Виконання програми завершено.");
    return 0;
}
void function1(int a)
{
printf("Ви ввели %d, тому була викликана function1\n\n", a);
}

void function2(int b)
{
printf("Ви ввели %d, тому була викликана function2\n\n", b);
}

void function3(int c)
{
    printf("Ви ввели %d, тому була викликана function3\ n\n", c);
}
```

Результат виконання програми наведено на рис. 6.28.

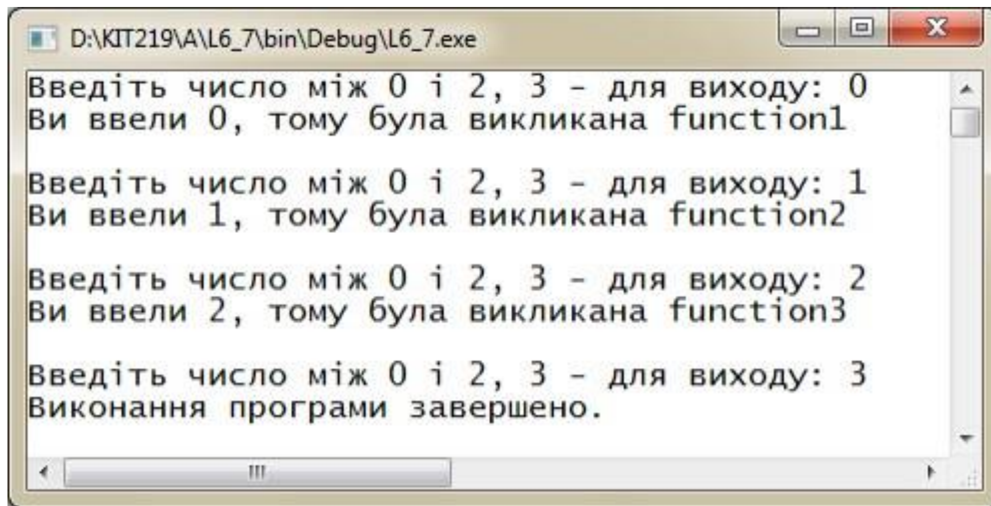


Рисунок 6.28 – Використання механізму вказівників на функції

Визначення масиву **f** читається, починаючи з самої лівої пари дужок: «**`f`** – це масив з трьома вказівниками на функції, кожна з яких приймає ціле число і нічого не повертає». Масив ініціалізується іменами трьох функцій. Коли користувач вводить значення в діапазоні від **`0`** до **`2`**, введене значення використовується в якості індексу в масиві вказівників на функції. У виклику функції **`f[choice]`** витягається вказівник на функцію з елемента з індексом **`choice`**. Потім вказівник розіменовується, щоб викликати функцію, при цьому функції передається значення **`choice`** в якості аргументу. Кожна функція виводить значення свого аргументу та ім’я функції, щоб показати, що виклик був виконаний правильно.

Контрольні запитання та завдання

1. Яким чином записується визначення функції?
2. Що таке прототип функції, коли він використовується.
3. Що таке тип функції? Які типи можна задавати для функції?
4. Чим виклик функції відрізняється від прототипу?
5. Що повертає функція, яка має тип **`void`**?
6. Яку роль виконують параметри функції?

7. Чим відрізняються формальні та фактичні параметри функції?

8. Розкажіть про використання змінних у функціях. Яка різниця між локальною та глобальною змінними?

9. Як передати масив до функції?

10. Для чого використовується ключове слово **`const`** при роботі з функціями?

11. Чому вказівник не може існувати як самостійний тип?

12. З якою метою в програмі може бути використаний вказівник типу **`void`?**

13. Що буде результатом розіменування вказівника типу **`void`** без приведення типів?

14. Як зміниться значення вказівника після застосування до нього операції інкременту (декременту)?

15. Чому для вказівників визначені додавання та віднімання тільки з цілими константами?

16. У чому відмінність вказівника на константу від вказівника-константи?

17. Два вказівника різних типів вказують на одне й те ж саме місце в пам'яті. Порівняйте результати операцій розіменування та взяття адреси з такими вказівниками. Порівняйте значення вказівників.

18. Якщо об'єкт займає в пам'яті кілька байтів, то яку адресу має значення вказівника на цей об'єкт?

Завдання для самостійного розв'язання

1. Дана квадратна матриця порядку M = 7. Напишіть *C*-програму, в якій функція, обчислює суму елементів її головної діагоналі. Знайдіть також суми елементів, що розташовані вище та нижче головної діагоналі. Обнуліть ті елементи матриці, сума яких виявилася найбільшою. Скористайтеся ініціалізацією матриці цілими числами від 10 до 90.

2. Напишіть *C*-програму, яка б циклічно виводила площу трукутникаа а потім пощу трапеції. Розрахунок кожної площі фігури слід проводити за допомогою окремих функцій. Необхідно також передбачити умову виходу з нескінченого циклу натисканням визначеної вами клавіші.

3. Напишіть С-програму у якій визначте та ініціалізуйте змінну типу **`double`**, вказівник на тип **`double`** і вказівник на тип **`void`**. Виконайте присвоєння вказівникам адреси змінної. Виведіть на консоль адресу змінної, значення вказівників і значення, які отримані після розіменування вказівників. Для демонстрації ролі та послідовності виконання унарних операцій отримання адреси `'&'` і розіменування `'*'`, виведіть на консоль значення виразу `"* & ім’я_змінної"`.

4. Напишіть *C*-програму, яка передбачає ввід з клавіатури натурального чотиризначного числа та обчислює в циклі суму всіх його цифр, використовуючи вказівники та операції над ними.

5. Визначте та ініціалізуйте в С-програмі змінну типу **`double`**. Визначте вказівники **`char *`**, **`int`** *, **`double`** *, **`void`** * та ініціалізуйте їх адресою змінної. Виведіть на консоль значення вказівників, їх розміри та довжини ділянок пам’яті, які пов’язані з виразами розіменування вказівників.

7. ФУНКЦІЇ ДЛЯ РОБОТИ З РЯДКАМИ

7.1. Рядки. Функції для вводу-виводу рядків

Символьний рядок є одним з найбільш корисних і важливих типів даних в мові C. До сих пір ми з вами постійно використовували символьні рядки, але вам треба буде ще багато нового дізнатися про них. Стандартна бібліотека C пропонує широкий спектр функцій для читання та запису, копіювання, порівняння, об'єднання, пошуку та виконання багатьох інших операцій з рядками. Сьогоднішня лекція допоможе вам поповнити свої навички програмування можливостями, які стосуються рядків і роботи з ними.

7.1.1. Рядки та ввід-вивід рядків

Відомо, що символьний рядок – це масив елементів типу **char**, що завершується нульовим символом ('\0'). Відповідно, все, що ви вже вивчили про масиви та вказівники, стосується і символьних рядків. Однак у зв'язку з широким використанням символьних рядків бібліотека C надає велику кількість функцій, які спеціально призначені для роботи з рядками. Розглянемо природу рядків, способи їх оголошення та ініціалізації, а також ввід і вивід рядків у програмах. На наступній лекції розглянемо різні маніпуляції з рядками.

Давайте спочатку розглянемо коротку програму, яка ілюструє декілька способів представлення рядків. Її текст має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#define MSG "Константа, яка є рядком символів."
#define MAXLENGTH 81

int main(void) {
    char words[MAXLENGTH] =
        "Рядок символів, що зберігається в масиві.";
    const char *pt1 = "Вказівник, що посилається на рядок.";

    SetConsoleOutputCP(1251);
    puts("Рядки символів:\n");
    puts(MSG);
```

```
    puts(words);
    puts(pt1);
    words[34] = 'п';
    puts(words);
    return 0;
}
```

Результат роботи програми наведено на рис. 7.1

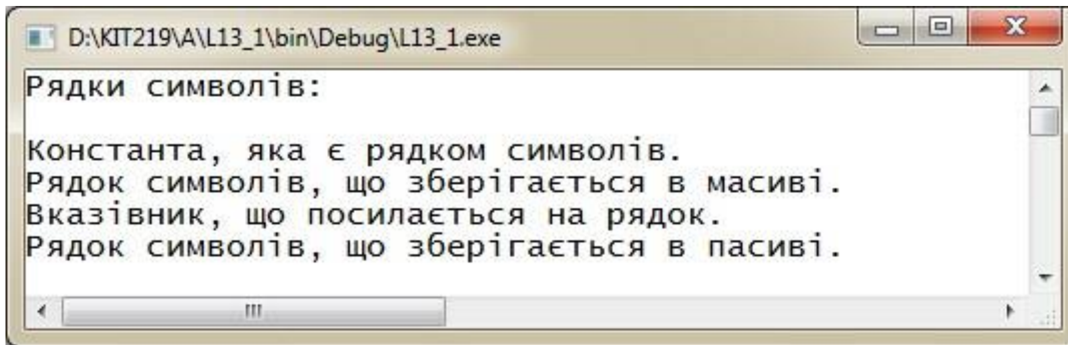


Рисунок 7.1. – Способи представлення рядків символів

Подібно до `printf()`, функція `puts()` належить до сімейства функцій вводу-виводу, для використання якої необхідно підключити файл заголовку `stdio.h`. Вона відображає тільки рядки і, на відміну від `printf()`, автоматично додає наприкінці символ нового рядка.

Розглянемо способи визначення рядка у програмі та з'ясуємо як відбувається його читання. Також спробуємо дослідити різні методи, які застосовуються для виводу рядків.

7.1.2. Визначення рядків у програмі

Існує багато способів визначення рядків. До основних з них слід віднести: використання константних рядків, масивів типу `char` і вказівників на тип `char`. Крім того, програма повинна забезпечити місце для зберігання рядка.

Символьні рядкові літерали (рядкові константи). Рядковий літерал, який також зветься рядковою константою, являє собою довільну послідовність символів, що розміщується в подвійних лапках. Символи, які розміщуються між одинарними лапками, а також завершальний символ `'\0'`, що автоматично додається компілятором, зберігаються у пам'яті як символьний рядок.

Починаючи зі стандарту **ANSI C**, виконується конкатенація (об'єднання) рядкових літералів, якщо вони відокремлені один від одного лише пробільними символами. Як це робиться, ви вже бачили в одній з попередніх лекцій, коли розглядалися рядки.

Символьні рядкові константи розміщуються в статичному класі зберігання, тобто, якщо ви використовуєте рядкову константу у функції, то цей рядок зберігається тільки один раз і існує протягом часу виконання програми, навіть якщо функція викликається багато разів. Уся фраза, яка розміщена в лапках, діє в якості вказівника на місце, де зберігається рядок. Це аналогічно імені масиву, яке трактується як вказівник на місце розміщення масиву.

*Масиви символьних рядків та їх ініціалізація.*Коли ви визначаєте масив символьних рядків, необхідно повідомити компілятор про те, скільки для нього слід виділити пам'яті. Один зі способів передбачає зазначення розміру масиву, який є достатнім для зберігання рядка. Наступне оголошення ініціалізує масив **m1** символами визначеного рядка:

```
const char m1[40]  = "Спробуйте вкластися в один рядок.";
```

Ключове слово **const** позначає намір не змінювати цей рядок.

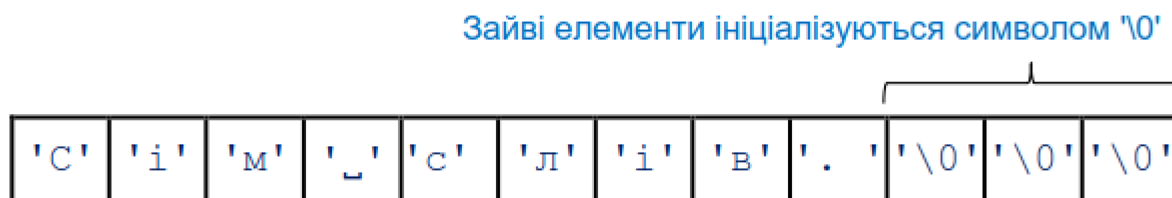
Вказана форма є скороченням для стандартної форми ініціалізації масиву:

```
const char m1[40] = { 'С', 'п', 'р', 'о', 'б', 'у',
                      'й', 'т', 'е', '␣', 'в', 'к',
                      'л', 'а', 'с', 'т', 'и', 'с',
                      'я', '␣', 'в', '␣', 'о', 'д',
                      'и', 'н', '␣', 'р', 'я', 'д',
                      'о', 'к', '.', '\0' };
```

Зверніть увагу на завершальний нульовий символ. Без нього ви отримаєте символьний масив, а не рядок.

При зазначенні розміру масиву слід переконатися, що кількість елементів, насамперед, на одиницю більше довжини рядка (не забувайте про нульовий символ). Будь-які невикористані елементи ініціалізуються значенням **'\0'** (яке являє собою нульовий символ у формі **char**).

На рис. 7.2 наведено приклад ініціалізації масиву символів у пам’яті комп’ютера.




```
const char sentence[12] = "Сім слів.";
```

Рисунок 7.2 – Приклад ініціалізації масиву

Часто зручно надавати компілятору право на визначення розміру масиву. Ви вже знаєте, що якщо розмір не вказано в оголошенні при ініціалізації масиву, компілятор самостійно визначить його розмір:

```
const char m2[] = "Довжина масиву визначається автоматично.";
```

Ініціалізація символьних масивів – це один з тих випадків, коли дійсно слід надати можливість компілятору самому визначити розмір масиву. Причина цього полягає в тому, що функції обробки масивів не потребують інформації про розмір масиву, оскільки вони можуть просто знаходити нульовий символ, який позначає кінець рядка.

Надання компілятору можливості обчислювати розмір масиву самостійно працює лише при ініціалізації масиву. Якщо ви створюєте масив, який слід заповнити пізніше, його розмір необхідно вказати в оголошенні.

Всередині оголошення розмір масиву повинен обчислюватися як цілочислове значення. До введення стандартом **C99** масивів змінної довжини (**variable length array – VLA**) розмір повинен був бути цілочисловою константою, що передбачає можливість застосування виразу, який утворений з константних цілочислових значень.

```
int n = 8;
char cookies[1];     // допустимо
char cakes[2 + 5];   // допустимо, оскільки розмір є
                     // константним виразом
char pies[2 * sizeof(long double) + 1]; // допустимо
char crumbs[n]; // в C99 це масив змінної довжини
```

Подібно до будь-якого іншого масиву, ім’я символьного масиву видає адресу першого елемента масиву. Відповідно, зазначене нижче твердження є справедливим:

```
char car[10]  = "Луна";
car == &car[0], *car == 'Л' і *(car + 1) == car[1] == 'у'
```

Дійсно, для ініціалізації рядка можна використовувати форму запису з вказівниками.

```
const char *pt1 = "Рядок символів.";
```

Це оголошення дуже близьке до такого оголошення:

```
const char ar1[] = "Рядок символів.";
```

Обидва оголошення говорять про те, що **pt1** і **ar1** є адресами рядків. В обох випадках рядок, позначений подвійними лапками, сам визначає необхідний об’єм пам’яті, який буде для нього зарезервований. Проте, ці форми не є ідентичними.

Масиви або вказівники. В чому полягає різниця між формами у вигляді масиву та вказівника? Запис у формі масиву **ar1[]** призводить до розміщення в пам’яті комп’ютера масиву з **16** елементів (по одному на кожен символ плюс один елемент для завершального символу **'\0'**). Кожен елемент ініціалізується відповідним символом рядкового літерала. Зазвичай рядок зберігається в сегменті даних, який є частиною виконавчого файлу. Коли програма завантажується в пам’ять, разом з нею завантажується і цей рядок. Говорять, що рядок знаходиться в статичній пам’яті. Однак пам’ять під масив виділяється тільки після того, як програма почне виконуватися. В цей час рядок копіюється до масиву. Зверніть увагу на те, що в цей момент існують дві копії рядка. Одна – це рядковий літерал в статичній пам’яті, а інша – рядок, що зберігається в масиві **ar1**.

В подальшому компілятор буде розпізнавати ім’я масиву **ar1** як синонім адреси першого елемента масиву, **&ar1[0]**. Один важливий аспект полягає в тому, що **ar1** в формі масиву є адресною константою. Значення

ar1 змінювати неможна, оскільки це означало б зміну місця (адреси), де зберігається масив. Для ідентифікації наступного елемента в масиві можна задіяти операції на зразок **ar1+1**, але вираз **++ar1** є неприпустимим. Операція інкремента може застосовуватися тільки до імен змінних (або, в загальному сенсі, до **l**-значень, що модифікуються), але не до констант.

Форма вказівника ***pt1** також призводить до того, що в статичній пам'яті під рядок резервується **16** елементів. Як тільки програма почне виконуватися, вона виділяє в пам'яті місце під змінну типу вказівника **pt1** і зберігає в ній адресу рядка. Спочатку ця змінна вказує на перший символ рядка, але її значення можна змінювати. Відповідно, можна використовувати операцію інкремента. Наприклад, **++pt1** буде вказувати на другий символ **'я'**.

Рядкові літерали вважаються даними **const**. Оскільки вказівник **pt1** посилається на такі дані, він повинен бути оголошений, як такий, що вказує на дані **const**. Це зовсім не означає, що неможна змінювати значення **pt1** (тобто місце, на яке він вказує), просто **pt1** не допускається застосовувати для зміни самих даних. З іншого боку, при копіюванні рядкового літералу до масиву, дані можна вільно змінювати, якщо тільки сам масив не був оголошений як **const**.

Іншими словами, ініціалізація масиву призводить до копіювання рядка зі статичної пам'яті до масиву, тоді як ініціалізація вказівника просто копіює адресу рядка. Програма, яка демонструє це, виглядає наступним чином:

```
#include <stdio.h>
#include <windows.h>
#define MSG "Я особлива."

int main(void)
{
    char ar[] = MSG;
    const char *pt = MSG;

    SetConsoleOutputCP(1251);

     printf("Адреса \"Я особлива.\": %p\n", "Я особлива.");
     printf(" адреса ar: %p\n", ar);
     printf(" адреса pt: %p\n", pt);
     printf(" адреса MSG: %p\n", MSG);
```

```
        printf("Адреса \"Я особлива.\": %p\n", "Я особлива.");
        return 0;
}
```

Результат виконання програми наведено на рис. 7.3.

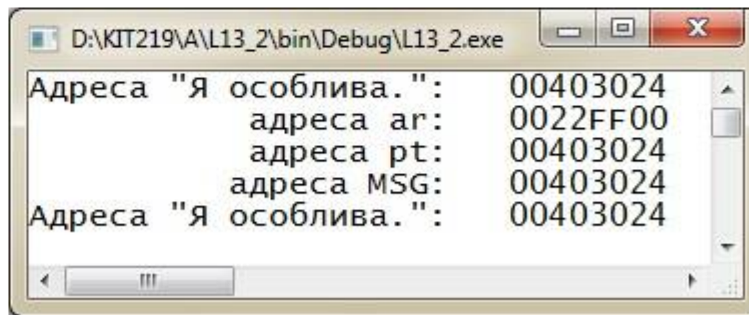


Рисунок 7.3 – Ініціалізація масиву та вказівника

Про що це свідчить? По-перше, `pt` і `MSG` – це одна й та ж сама адреса, але `ar` є іншою адресою. По-друге, літерал `"Я особлива."` зустрічається в операторах `printf()` двічі. Компілятор використовує одну область пам’яті з однією адресою. Йому надається свобода вибору зберігати літерал, який застосовується більше одного разу, в одному або декількох місцях.

Чи є важливою різниця в представленні рядка у вигляді масиву або вказівника? Частіше за все ні, однак це залежить від того, що саме ви намагаєтесь робити.

Різниця між масивами та вказівниками. Давайте розглянемо різницю між ініціалізацією символьного масиву, який призначений для зберігання рядка, та ініціалізацією вказівника, який вказує на цей рядок. Наприклад, розглянемо наступні два оголошення:

```
char heart[] = "Константа";
const char *head = "Вказівник";
```

Головна різниця між ними полягає в тому, що ім’я масиву `heart` є константою, а вказівник `head` – змінною. Що це означає на практиці? Перш за все, в обох випадках можна застосовувати форму запису з масивом:

```
for(int i = 0; i < 9; i++)
    putchar(heart[i]);
putchar('\n');
```

```
for(int i = 0; i < 9; i++)
    putchar(head[i]);
putchar('\n');
```

На консолі отримаємо наступне (рис. 7.4):

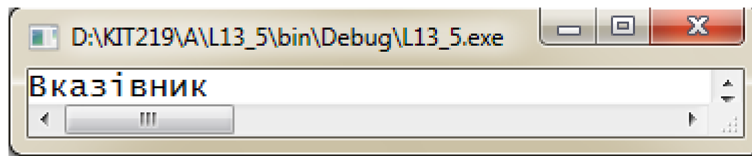


Рисунок 7.4 – Використання форми запису з масивом

Також в обох випадках можна використовувати додавання значення до вказівника:

```
for(int i = 0; i < 9; i++)
   putchar(*(heart + i));
putchar('\n');
for(int i = 0; i < 9; i++)
   putchar(*(head + i));
putchar('\n');
```

І знову отримаємо те ж саме на консолі (рис. 7.5):

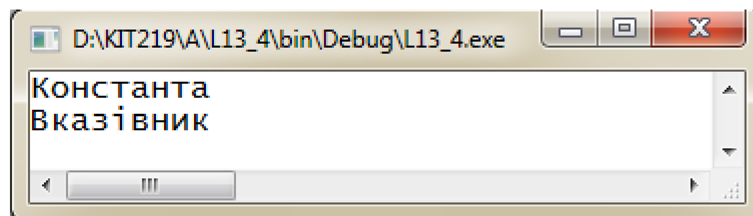


Рисунок 7.5 – Додавання значення до вказівника

Однак операція інкремента може застосовуватися лише у версії з вказівником:

```
while(*(head) != '\0') // залишитися наприкінці рядка

putchar(*(head++));  // вивести символ, перемістити вказівник
```

Цей код дає на консолі наступне (рис. 7.6):

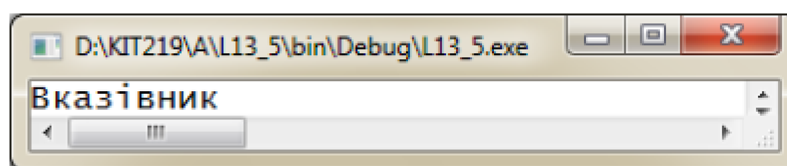


Рисунок 7.6 – Результат застосування операції інкремента для вказівника

Не використовуйте вказівник на рядковий літерал, якщо ви плануєте змінювати рядок.

7.1.3. Масиви символьних рядків

Часто зручно мати масив символьних рядків. Тоді для доступу до різних рядків можна застосовувати індекс. В наступній програмі продемонстровані два підходи: масив вказівників на рядки та масив, який складається з масивів типу **char**. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#define SLEN 40
#define LIM 5

int main(void)
{
     const char *mytalents[LIM] =
     {
             "Миттєве додавання чисел",
             "Точне обчислення добутку",
             "Накопичування даних",
             "Виконання інструкцій з точністю до букви",
             "Знання мови програмування C"
     };

         char yourtalents[LIM][SLEN] =
         {
             "Ходіння по прямій",
             "Здоровий сон",
             "Перегляд телепередач",
             "Розсилка листів",
             "Читання електронної пошти"
         };

     SetConsoleOutputCP(1251);

     puts("Порівняємо наші таланти.");
     printf("%-52s %-25s\n", "Мої таланти", "Ваші таланти");
     for(int i = 0; i < LIM; i++)
         printf("%-52s %-25s\n", mytalents[i], yourtalents[i]);
     printf("\n\nРозмір mytalents: %d, розмір yourtalents: %d\n\n",
         sizeof(mytalents), sizeof(yourtalents));
     return 0;
}
```

Результат роботи програми наведено на рис. 7.7.

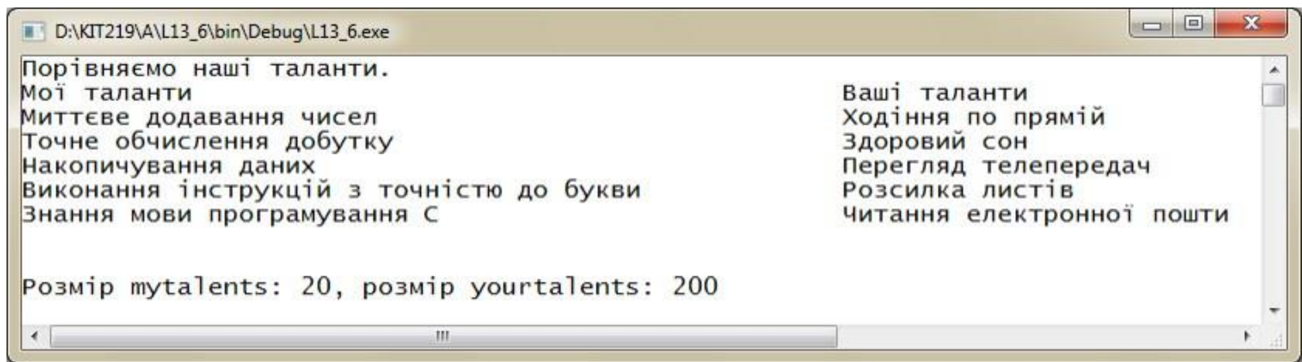


Рисунок 7.7 – Приклад роботи з символьними масивами

Багато в чому масиви **mytalents** і **yourtalents** дуже схожі. Кожен з них представлений п'ятьма рядками. Коли використовується один індекс (**mytalents[0]** і **yourtalents[0]**), результатом буде одиночний рядок. Обидва масиви ініціалізуються однаковою мірою.

Але є і розбіжності. Масив **mytalents** – це масив з п'яти вказівників, який займає в системі **20** байтів. Масив **yourtalents** складається з п'яти масивів по **40** значень типу **char** і займає в системі **200** байтів. Таким чином, тип масиву **mytalents** відрізняється від типу **yourtalents**, незважаючи на те, що і **mytalents[0]**, і **yourtalents[0]** – це рядки. Вказівники в **mytalents** вказують на місця розміщення рядкових літералів, що застосовуються для ініціалізації, які зберігаються в статичній пам'яті. Однак масиви в **yourtalents** містять копії рядкових літералів, в результаті чого кожен рядок зберігається двічі.

Більш того, розподіл пам'яті в масивах є неефективним, оскільки усі елементи **yourtalents** повинні мати однаковий розмір, і цей розмір повинен бути достатньо великим, щоб вмістити найдовший рядок.

Одним зі способів сприйняття цієї розбіжності є представлення **yourtalents** у вигляді прямокутного двовимірного масиву, усі рядки якого мають однакову довжину (в даному випадку **20** байтів). Разом з тим **mytalents** можна представити у вигляді зубчатого масиву зі змінюваною

довжиною рядків. Ці два види масивів показані на рис. 7.8.

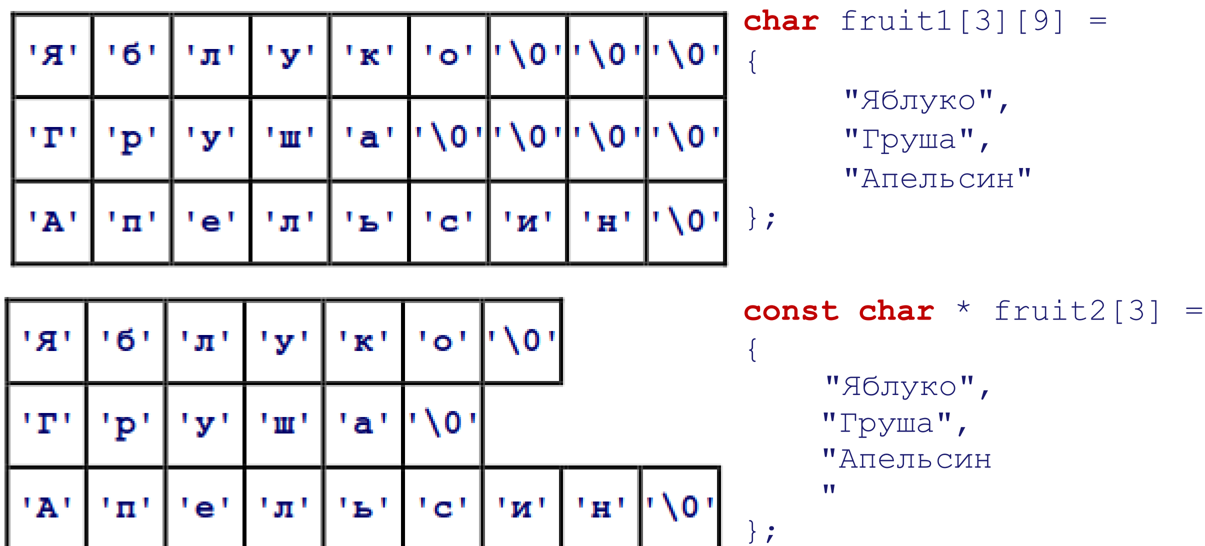


Рисунок 7.8 – Прямокутні та зубчаті масиви (відмінності в оголошенні)

7.1.4. Вказівники та рядки

Більшість операцій з рядками в `C` фактично працюють з вказівниками. Як приклад, розглянемо коротку програму, текст якої має такий вигляд:

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

int main(void)
{
    const char *mesg = "Вивчайте мову C!";
    const char *copy;

    SetConsoleOutputCP(1251);

    copy = mesg;
    printf("%s\n", copy);
printf("mesg = %s;\n&mesg = %p;\nvalue = %p\n\n",

                mesg, &mesg, mesg);
printf("copy = %s;\n&copy = %p;\nvalue = %p\n\n",
                copy, &copy, copy);
    return 0;
}
```

Дивлячись на цю програму, можна подумати, що вона створює копію рядка `"Вивчайте мову C!"`, і, на перший погляд, результат на консолі підтверджує це передбачення (рис. 7.9):

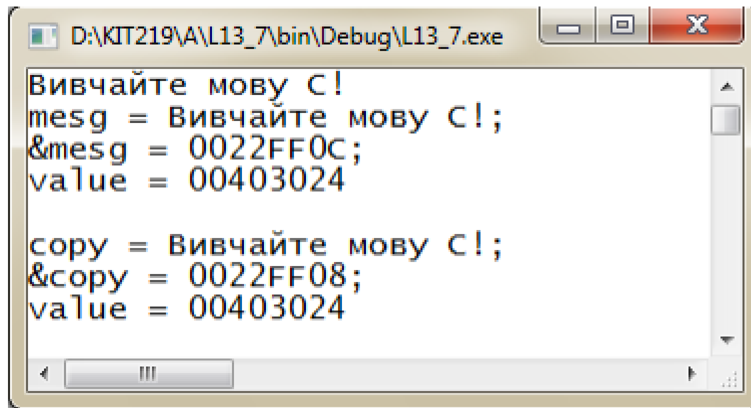


Рисунок 7.9 – Операції з рядками

Але давайте поглянемо на те, що виводить функція **printf()** більш уважно. Перш за все, значення **mesg** і **copy** виводяться у вигляді рядка (%s). Усі рядки виглядають як "Вивчайте мову C!".

Наступний елемент в кожному рядку являє собою адресу певного вказівника. Для конкретного запуску два вказівника **mesg** і **copy** зберігаються, відповідно, в комірках **0x0022ff0C** і **0x0022ff08**.

Тепер зверніть увагу на завершальний елемент на ім'я **value**. Це значення заданого вказівника. Значенням вказівника є адреса, яку він містить. Як можна побачити, **mesg** вказує на комірку **0x00403024**, і те ж саме можна сказати про **copy**. Відповідно, сам рядок не копіювався. Оператор **copy = mesg;** лише створює другий вказівник, який посилається на той самий рядок.

Тоді навіщо всі ці обережності? Чому б просто не скопіювати весь рядок? Задайте собі питання: що є більш ефективним – копіювати одну адресу або, скажімо, 50 окремих елементів? Часто для рішення задачі достатньо тільки адреси. Якщо вам дійсно необхідна копія рядка, тобто її дублікат, можна скористатися функцією **strcpy()** або **strncpy()**, як буде показано пізніше.

Тепер розглянемо рядки, які вводяться з клавіатури.

7.1.5. Функції для вводу рядків

Якщо в програмі треба ввести рядок символів, знадобиться спочатку

зарезервувати місце для зберігання цього рядка, а потім за допомогою функції вводу отримати цей рядок.

Створення вільного місця для рядка. Спочатку необхідно підготувати місце для розміщення рядка після його читання. Як згадувалося раніше, це означає виділення необхідного простору, який був би достатнім для подальшого читання рядків. Не слід очікувати, що комп'ютер визначить довжину рядка під час його читання і надасть необхідний простір. Комп'ютер не буде робити це (якщо тільки ви не напишете спеціальну функцію). Наприклад, припустимо, що існує такий код:

```
char *name;
scanf("%s", name);
```

Можливо, що компілятор прийме цей код, скоріше за все, з попередженням, але при читанні **name** рядок буде записаний поверх даних або коду вашої програми та аварійно її завершить. Причина полягає в тому, що функція **scanf()** копіює інформацію за адресою, яка надається аргументом, а в цьому випадку аргументом є неініціалізований вказівник (**name** може вказувати куди завгодно). Більшість програмістів находять таку ситуацію досить кумедною, але тільки не у власних програмах.

Простіше за все вирішити цю проблему, включивши в оголошення явний розмір масиву:

```
char name[81];
```

Тепер **name** являє собою адресу зарезервованого блока пам'яті розміром **81** байт. Інша можливість передбачає використання функцій виділення пам'яті з бібліотеки **C**.

Після надання місця для рядка його можна прочитати. Бібліотека **C** пропонує три функції, які дозволяють зчитувати рядки: **scanf()**, **gets()** і **fgets()**. Частіше за інші використовується функція **gets()**, тому розглянемо її першою.

Недоліки функції ***gets().*** Згадайте, що при читанні рядка функція

scanf() і специфікатор **%s** забезпечують зчитування тільки одного слова. Однак часто бажано, щоб програма могла зчитати одразу весь рядок, а не одне слово. Саме таку ціль переслідувала протягом багатьох років функція **gets()**. Це досить проста та зручна в користуванні функція. Вона читає весь рядок до символу нового рядка, відкидає цей символ і зберігає інші символи, додаючи нульовий символ, щоб створити рядок. Часто ця функція застосовується у поєднанні з функцією **puts()**, яка відображає рядок з додаванням символу нового рядка.

Розглянемо приклад програми, яка наводить можливість застосування функцій **gets()** і **puts()**. Її текст має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#define STLEN 81

 int main(void)
 {
     char words[STLEN];

     SetConsoleCP(1251);
     SetConsoleOutputCP(1251);

     puts("Введіть рядок.");
     gets(words);
     printf("\n\nВаш рядок виводиться двічі:\n");
     printf("%s\n", words);
     puts(words);
     puts("Готово.\n\n");
     return 0;
 }
```

Результат роботи програми наведено на рис. 7.10.

Зверніть увагу на те, що весь введений рядок крім символу нового рядка зберігається в масиві **words**, а виклик **puts(**words**)** забезпечує такий самий ефект, як і виклик **printf(**"%s\n", words**)**.

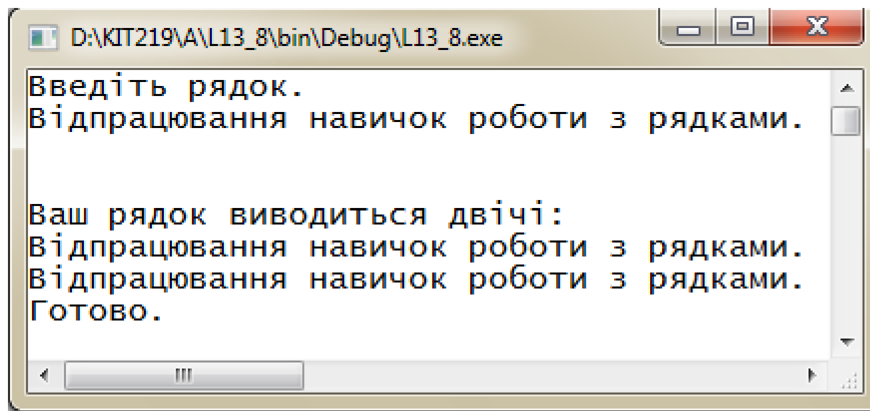


Рисунок 7.10 – Використання функцій **gets()** і **puts()**

Якщо рядок, що вводиться, буде занадто великим, виникне переповнення буфера іншими словами. Символи переповнять простір, який був призначений для них. Зайві символи можуть потрапити до пам'яті, що не використовується, і призвести до проблем, які з'являться не одразу, або ж вони можуть перезаписати інші дані в програмі. І це ще не усі можливі варіанти.

Чому ж тоді функції **gets()** приділяється особлива увага? Мабуть тому, що її непередбачувана поведінка створює ризик для безпеки. В минулому зловмисники користувалися особливостями системних програм, в яких використовувалася функція **gets()**, для вставляння і виконання коду, який компрометував безпеку системи.

На певному етапі багато представників спільноти програмістів на **C** рекомендували виключити функцію **gets()** з програмного словника. Комітет, що створив стандарт **C99**, також опублікував обґрунтування стандарту. В ньому були визнані проблеми, які пов'язані з використанням функції **gets()**, і застосовувати її не рекомендувалося. Разом з тим в обґрунтуванні було виправдано збереження функції **gets()** в якості частини стандарту через її зручність у випадку коректного використання, а також через те, що вона була частиною величезного об'єму існуючого коду.

Проте, комітет зі створення стандарту **C11** дотримувався більш суворих поглядів і виключив функцію **gets()** зі стандарту. З іншого боку, стандарт визначає те, що компілятор повинен підтримувати, а не те, що він

не повинен підтримувати. На практиці більшість компіляторів продовжують підтримувати цю функцію задля зворотної сумісності.

Альтернативи функції `gets()`. Традиційною альтернативою **`gets()`** є функція **`fgets()`**, яка має дещо більш складний інтерфейс і трохи по-іншому обробляє введені дані. Крім того, в стандарті **`C11`** до загального набору додана функція **`gets_s()`**. Вона більш схожа на **`gets()`** і її легше використовувати в існуючому коді в якості заміни. Проте, вона є частиною необов'язкового розширення сімейства функцій вводу-виводу **`stdio.h`**, і тому компілятори стандарту **`C11`** не зобов'язані її підтримувати.

Функція **`fgets()`** запобігає появі можливої проблеми переповнення, приймаючи другий аргумент, який обмежує кількість зчитуваних символів. Ця функція призначена для файлового вводу, що трохи ускладнює її застосування. Нижче перераховані її відмінності від функції **`gets()`**.

- вона приймає другий аргумент, який задає максимальну кількість символів для читання. Якщо цей аргумент має значення **`n`**, то функція **`fgets()`** прочитає **`n-1`** символів або буде читати рядок до появи символу нового рядка в залежності від того, що відбудеться раніше;

- якщо функція **`fgets()`** стикається з символом нового рядка, вона зберігає його в рядку, на відміну від функції **`gets()`**, яка відкидає його;

- функція **`fgets()`** приймає третій аргумент, який вказує файл, з якого необхідно проводити зчитування. Для зчитування з клавіатури в якості цього аргументу використовується **`stdin`** (від **`standard input`** – стандартний ввід). Цей ідентифікатор визначений у файлі заголовку **`stdio.h`**.

Оскільки функція **`fgets()`** обробляє символ нового рядка як частину рядка (за умови, що рядок вводу має відповідну довжину), її часто застосовують спільно з функцією **`fputs()`**, яка працює подібно **`puts()`**, але не додає автоматично символ нового рядка. Функція **`fputs()`** приймає другий аргумент, що вказує на файл, в який повинен проводитися запис. Для

виводу на консоль можна використовувати аргумент **stdout** (від **standard output** – стандартний вивід).

Розглянемо програму, в якій ілюструється поведінка функцій **fgets()** і **fputs()**. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#define STLEN 15

int main(void)
{
    char words[STLEN];

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    puts("Введіть рядок.");
    fgets(words, STLEN, stdin);
    printf("Ваш рядок виводиться двічі (за допомогою puts(),"

           " a потім fputs()):\n");
    puts(words);
    fputs(words, stdout);
    puts("\nВведіть ще один рядок.");
    fgets(words, STLEN, stdin);
    printf("Ваш рядок виводиться двічі (за допомогою puts(),"
           " a потім fputs()):\n");
    puts(words);
    fputs(words, stdout);
    puts("\n\nГотово.");
    return 0;
}
```

Результат роботи програми наведено на рис. 7.11.

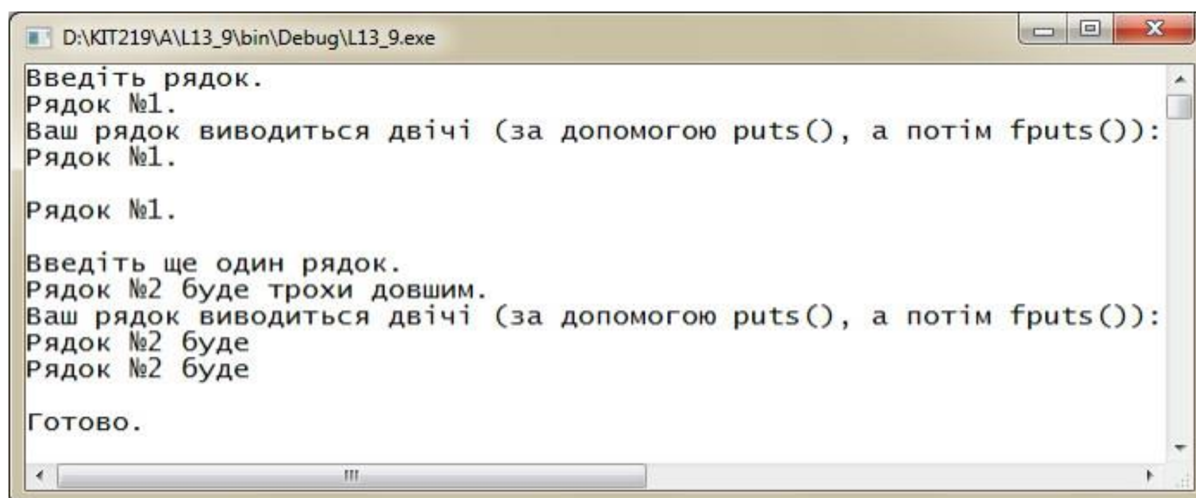


Рисунок 7.11 – Використання функцій **gets()** і **puts()**

Перший рядок вводу "Рядок №1." є досить коротким, щоб функція **fgets()** прочитала його та зберегла "Рядок №1.\n\0" в масиві. Тому, коли функція **puts()** відображає рядок і додає до виводу власний символ нового рядка, вона породжує порожній рядок виводу після рядка "Рядок №1.". Оскільки **fputs()** не додає символ нового рядка, вона не створить порожній рядок.

Довжина другого рядка вводу "Рядок №2 буде трохи довшим." перевищує ліміт на розмір (#define **STLEN** 15), тому **fgets()** зчитує перші **14** символів і зберігає в масиві рядок "Рядок №2 буде \0". В даному випадку функція **puts()** не породжує порожній рядок, і **fputs()** також не робить цього.

Функція **fgets()** повертає вказівник на тип **char**. Якщо все відбувається нормально, вона просто повертає ту саму адресу, яка була їй передана в першому аргументі. Однак якщо функція зустрічає кінець файлу, вона повертає спеціальний вказівник (нульовий). Такий вказівник гарантовано не посилається на реальні дані, тому може використовуватися для відображення особливого випадку. В коді він може бути представлений цифрою **0** або, що більш поширено в **C**, макросом **NULL**. Функція повертає **NULL** також в ситуації, коли виникла будь-яка помилка читання.

Наступна програма демонструє простий цикл, який читає і виводить на консоль текст до тих пір, поки функція **fgets()** не зустріне кінець файлу або не виконає зчитування порожнього рядка – тобто рядка, першим символом якого є символ нового рядка (тобто необхідно натиснути клавішу **<ENTER>**).

```
#include <stdio.h>
#include <windows.h>
#define STLEN 10

int main(void)
{
    char words[STLEN];

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
```

```
    puts("Введіть рядки (або порожній рядок для виходу з"
         " програми):");
    while(fgets(words, STLEN, stdin) != NULL
                 && words[0] != '\n')
        fputs(words, stdout);
    puts("Готово.");
    return 0;
}
```

Результат виконання програми наведено на рис. 7.12.

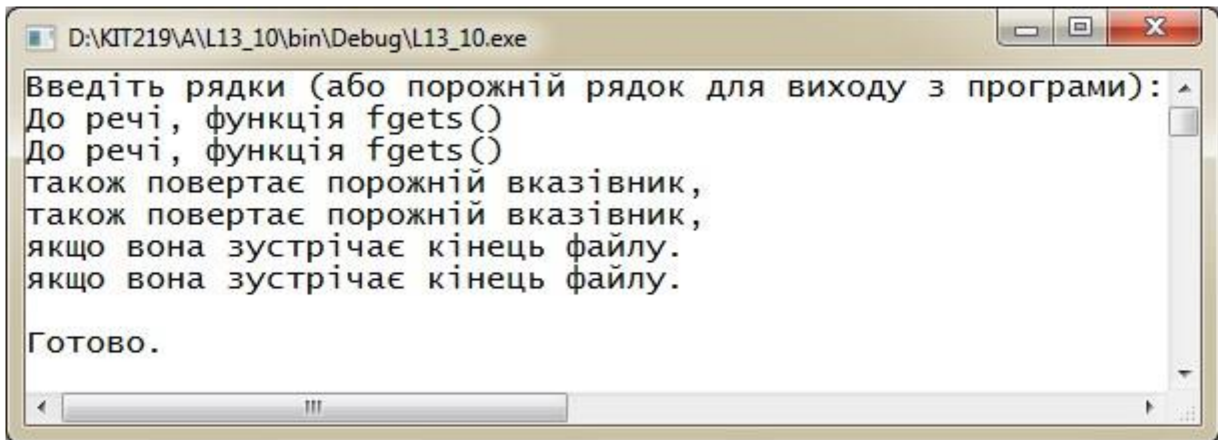


Рисунок 7.12 – Використання функцій **fgets()** і **fputs()**

Слід відзначити, що хоча значення **STLEN** дорівнює **10**, схоже, програма не має проблем під час обробки рядків вводу, довжина яких значно перевищує цей ліміт. Справа в тому, що в даній програмі функція **fgets()** читає **STLEN-1** (тобто **9**) символів за один раз. Тому вона починає з читання рядка "До речі, ", зберігає його як "До речі, \0". Потім **fputs()** відображає цей рядок, але при цьому не переходить до наступного рядку виводу. Далі функція **fgets()** поновлює читання з того місця, де вона зупинилася, і зчитує "функція f", зберігаючи її як "функція f\0". Функція **fputs()** відображає цей рядок в тому ж самому рядку, в якому вона знаходилася раніше. Потім **fgets()** поновлює читання вводу, продовжуючи до тих пір, поки не залишиться прочитати тільки "gets()\n". Функція **fgets()** зберігає рядок "gets()\n\0", функція **fputs()** відображає його, а внутрішній символ нового рядка призводить до переміщення курсору

на наступний рядок.

В системі використовується буферизований ввід-вивід. Це означає, що введені дані зберігаються в тимчасовій пам’яті (буфері) до тих пір, поки не буде натиснута клавіша **<ENTER>**. В результаті цього до введених даних додається символ нового рядка, і весь рядок передається функції **fgets()**. Під час виводу, функція **fputs()** передає символи до іншого буфера, і, після відправлення символу нового рядка, вміст буфера передається на консоль.

Той факт, що функція **fgets()** зберігає символ нового рядка, породжує проблему, але й надає додаткову можливість. Проблема полягає в тому, що зберігання символу нового рядка у вигляді частини рядка може бути небажаним. Перевага полягає в тому, що наявність або відсутність символу нового рядка у збереженому рядку може бути індикатором того, чи був прочитаний весь рядок. Якщо цього символу немає, можна приймати рішення про те, що робити з тією частиною рядка, що залишилася.

По-перше, як позбавитися від символу нового рядка? Один зі способів – його пошук в збереженому рядку і заміна нульовим символом:

```
while(words[i] != '\n')  // передбачається, що \n є в
    words i++;
words[i] = '\0';
```

По-друге, як бути, якщо в рядку вводу як і надалі залишаються символи? Розумний підхід на випадок, якщо увесь рядок не поміщається в цільовому масиві, передбачає ігнорування тієї частини, яка не вміщується:

```
while(getchar() != '\n') // читання без збереження
continue;                // вводу, включаючи \n
```

В наступній програмі до цих базових ідей додається додаткова перевірка, що дає у підсумку код, який читає рядки вводу, видаляє збережені символи нового рядка, якщо такі є, і відкидає частину рядка, яка не вміщується у виділену область пам’яті. Текст програми має такий вигляд:

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

```
#include <windows.h>
#define STLEN 10

int main(void)
{
   char words[STLEN];
   int i;

   SetConsoleCP(1251);
   SetConsoleOutputCP(1251);

   puts("Введіть рядки (або порожній рядок для виходу з"
        " програми):");
   while(fgets(words,STLEN,stdin) != NULL && words[0] != '\n')
   {
         i = 0;
         while(words[i] != '\n' && words[i] != '\0')
            i++;
         if(words[i] == '\n')
             words[i] = '\0';
         else         // необхідна наявність words[i] == '\0'
         while(getchar() != '\n')
             continue;
         puts(words);
    }
    puts("Готово.");
    return 0;
}
```

Результат виконання програми наведено на рис. 7.13.

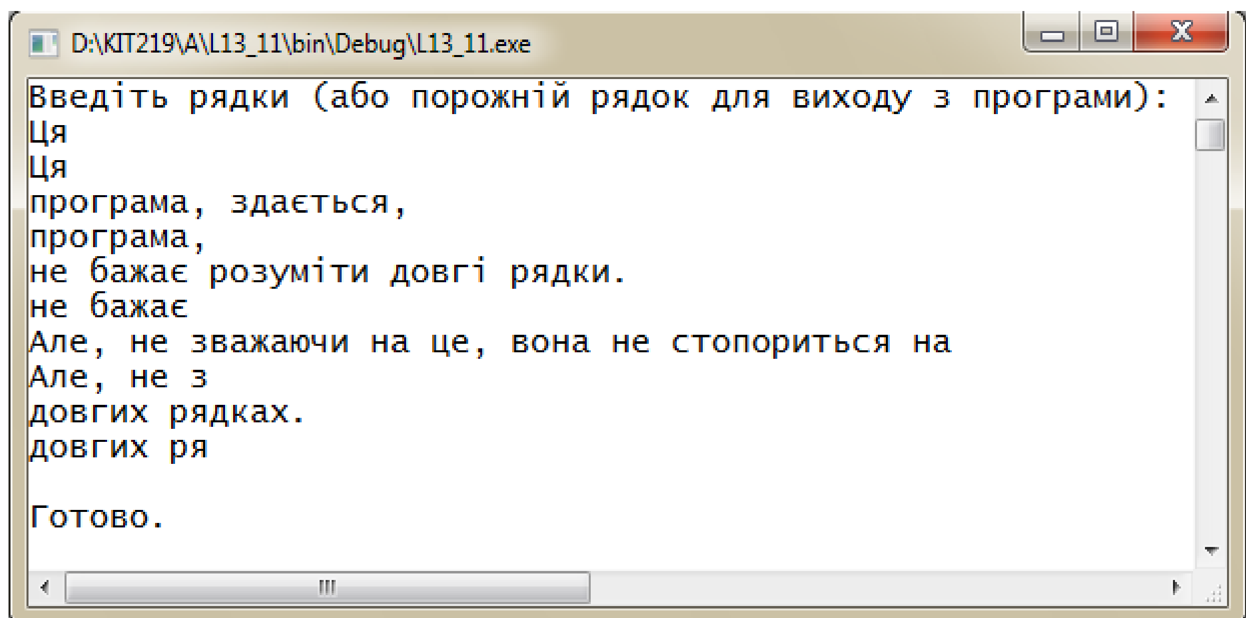


Рисунок 7.13 – Використання функцій **fgets()** і **fputs()** з відсіканням частини рядка

Цикл.

```
while(words[i] != '\n' && words[i] != '\0')
   i++;
```

виконує прохід по рядку до тих пір, поки не зустріне символ нового рядка або нульовий символ (в залежності від того, що відбудеться раніше). Якщо знайдений символ є символом нового рядка, то наступний за циклом оператор `if` замінює його нульовим символом. В іншому випадку – частина `else` відкидає залишок рядка вводу.

Нульовий символ і нульовий вказівник. У попередній програмі були присутні і нульовий символ, і нульовий вказівник. Концептуально ці дві «нульові» сутності відрізняються одна від одної. Нульовий символ, або `'\0'`, є символом, який використовується для ознаки кінця рядку `C`. Цей символ має код, який дорівнює нулю. Оскільки даний код не відповідає жодному іншому символу, нульовий символ не може випадково з'явитися в будь-якій іншій частині рядка.

Нульовий вказівник, або `NULL`, має значення, яке не відповідає будь-якій допустимій адресі даних. Він часто використовується у функціях, які в іншому випадку повертають допустимі адреси, для вказування на деяку спеціальну ситуацію, таку як виявлення ознаки кінця файлу або неможливість виконання дії яка очікувалася.

Таким чином, нульовий символ має цілочисловий тип, а нульовий вказівник має тип вказівника.

Іноді плутанина виникає через те, що числовим представленням їх обох може виявитися значення `0`. Але концептуально – це різні типи `0`. Крім того, нульовий символ, будучи символом, займає один байт, тоді як нульовий вказівник, як адреса, зазвичай займає чотири байти.

Функція `gets_s()`. Необов'язкова функція `gets_s()` в `C`, подібно `fgets()`, застосовує аргумент для обмеження кількості зчитаних символів. З урахуванням визначень попередньої програми, наступний код буде зчитувати рядок вводу до масиву `words`, забезпечуючи появу символу нового рядка в числі перших `9` символів вводу:

```
gets_s(words, STLEN);
```

Нижче описані три основні відмінності цієї функції від `fgets()`.

1) Функція `gets_s()` просто виконує зчитування зі стандартного вводу, тому вона не потребує третього аргументу.

2) Якщо функція `gets_s()` зчитує символ нового рядка, то відкидає його, а не зберігає.

3) Якщо `gets_s()` прочитає максимальну кількість символів, і серед них символ нового рядка буде відсутній, вона виконає декілька дій. Функція встановлює перший символ цільового масиву в нульовий символ. Потім вона читає і відкидає наступні символи, які були введені, поки не зустрінеться символ нового рядка або ознака кінця файлу. Нарешті, функція повертає нульовий вказівник. Вона викликає функцію «обробника», яка залежить від реалізації, (або ж обрану вами функцію), яка може привести до виходу з програми або припиненню її роботи.

Друга відмінність означає, що до тих пір, поки рядок вводу не надто довгий, функція `gets_s()` веде себе подібно `gets()`, завдяки чому `gets()` простіше замінити функцією `gets_s()`, ніж `fgets()`. Третя відмінність означає необхідність в навчанні стосовно використання цієї функції.

Давайте порівняємо прийнятність функцій `gets()`, `fgets()` і `gets_s()`. Якщо рядок вводу вміщується в цільову область пам'яті, то усі три функції працюють успішно. Але функція `fgets()` приєднує до рядка символ нового рядка, тому може виникнути необхідність в коді для його заміни нульовим символом.

А що буде відбуватися, якщо довжина рядка вводу буде перевищувати заданий розмір? Тоді використання функції `gets()` буде небезпечним – можуть бути пошкоджені дані та порушена безпека. Функція `gets_s()` безпечна, але якщо ви не бажаєте, щоб програма припинила роботу або виконала вихід будь-яким способом, доведеться подумати над тим, як написати та зареєструвати спеціальні «обробники». Крім того, якщо все же

вдається зберегти програму у працездатному стані, функція **gets_s()** відкине частину рядка вводу, що залишилася, незалежно від вашого бажання. У випадку, коли рядок не вміщується в задану область пам'яті, з функцією **fgets()** мати справу простіше, ніж з двома іншими функціями, і вона надає більше можливих варіантів.

Таким чином, коли ввід не відповідає очікуванням, функція **gets_s()** є менш зручною і гнучкою у порівнянні з **fgets()**. Ймовірно, це одна з причин, за якою функція **gets_s()** являє собою лише необов'язкове розширення бібліотеки C. З урахуванням необов'язковості **gets_s()** зазвичай кращим вибором буде функція **fgets()**.

Функція s_gets(). В попередній програмі був представлений один із способів застосування функції **fgets()**: читання всього рядка і заміна символу нового рядка нульовим символом або читання частини рядка, яка вміщується в задану область пам'яті, і відкидання решти символів – тобто свого роду різновид функції **gets_s()**, але без додаткових перешкод. Жодна зі стандартних функцій не задовольняє цьому опису, але ми можемо створити таку функцію самостійно. Вона буде в нагоді в майбутньому. Наступна функція демонструє один з підходів. Її текст буде мати такий вигляд:

```
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                   // треба наявність words[i] == '\0'
      while(getchar() != '\n')
          continue;
   }
return ret_val;
}
```

В той час як функція fgets() повертає NULL, вказуючи на кінець файлу або помилку читання, функція s_gets() пропускає обробку решти даних. В іншому випадку вона імітує роботу попередньої програми, замінюючи в рядку символ нового рядка нульовим символом, якщо даний символ присутній, і, відкидає частину рядка, що залишилася, в іншій ситуації. Потім вона повертає те ж саме значення, що й fgets(). Будемо використовувати цю функцію в подальшому.

В чому сенс відкидання частини, що залишилася, для надто довгого рядка. Проблема полягає в тому, що якщо її залишити, вона стане вхідними даними для наступного оператора читання. Це може, наприклад, спричинити аварійне завершення програми, якщо наступний оператор читання очікує значення типу double. Відкидання частини рядка, що залишилася, підтримує оператори читання у стані синхронізації з вводом з клавіатури.

Функція s_gets() не є ідеальною. Найбільш серйозний її недолік полягає в тому, що вона відкидає зайві вхідні дані, не повідомляючи про це ні програму, ні користувача, і тим самим позбавляючи його інших можливостей, таких як повторення вводу або знаходження більшого об'єму пам'яті. Ще один недолік – відсутність заходів на випадок неправильного застосування на кшталт передачі розміру, який дорівнює 1 або меншого за 1. Не зважаючи на це, дана функція цілком може бути заміною функції gets() в наших прикладах.

Функція scanf(. Давайте знову повернемося до функції scanf(). Раніше для читання рядка ми використовували функцію scanf() з форматом %s. Основна відмінність між функціями scanf() і fgets() пов'язана з тим, як вони визначають момент досягнення кінця рядка: функція scanf() більше орієнтована на «отримання слова», а не на «отримання рядка». Функція gets() приймає усі символи аж до першого символу нового рядка, як це робить fgets(), якщо рядок є достатньо коротким.

Функція **scanf()** має у своєму розпорядженні дві можливості для припинення вводу. При будь-якому варіанті рядок починається з першого непробільного символу, який зустрінеться. Якщо заданий формат %s, рядок продовжується до наступного (не включаючи його) пробільного символу (символу пробілу, табуляції або нового рядка). Якщо вказана ширина поля, наприклад, %10s, функція **scanf()** читає до отримання **10** символів або до появи першого пробільного символу, в залежності від того, що відбудеться раніше (табл. 7.1). Згадайте, що функція **scanf()** повертає цілочислове значення, яке дорівнює кількості успішно прочитаних елементів, або **EOF** при виявленні кінця файлу.

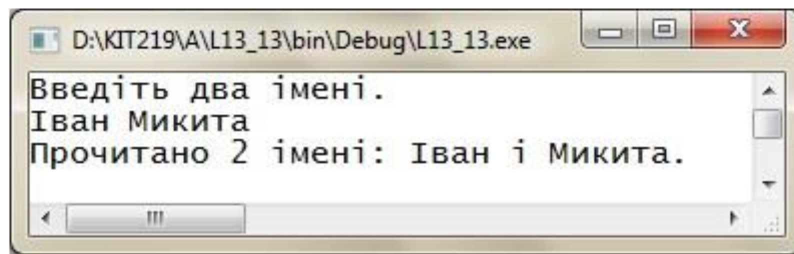
Таблиця 7.1 – Ширина поля та функція **scanf()**

Оператори вводу	Початкова	Вміст рядка імені	Частина черги, що залишилася
scanf("%s", name);	Іванов␣Іван	Іванов	␣Іван
scanf("%5s", name);	Іванов␣Іван	Івано	в␣Іван
scanf("%5s", name);	Іван␣Іванов	Іван	␣Іванов

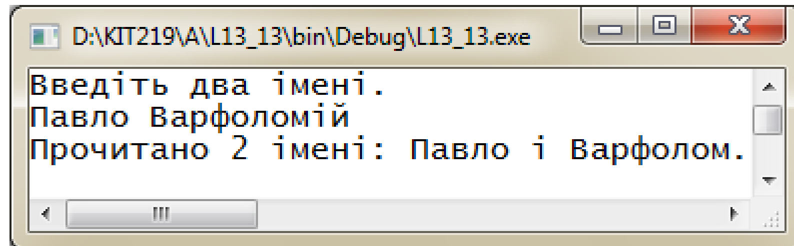
В наступній програмі ілюструється робота функції **scanf()**, коли задана ширина поля. Текст програми має такий вигляд:

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

int main(void)
{
    char name1[11], name2[11];
    int count;

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    printf("Введіть два імені.\n");
    count = scanf("%5s %8s", name1, name2);
    printf("Прочитано %d імені: %s і %s.\n",
        count, name1, name2);
    return 0;
}
```

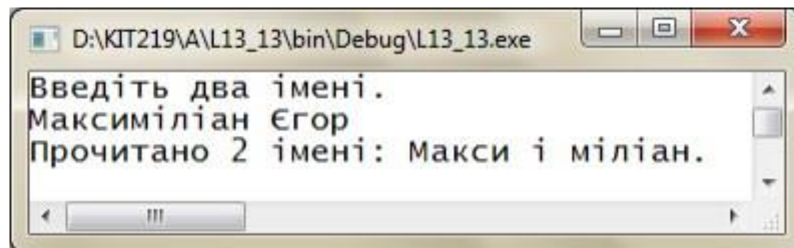
Результат роботи програми наведено на рис. 7.14.



а



б



в

Рисунок 7.14 – Використання функції **scanf()**

Під час першого запуску (рис. 7.14, *а*) обидва імені вписуються в дозволені граничні розміри. Під час другого запуску були прочитані тільки перші **8** символів імені **Варфолом**. оскільки ми застосовуємо формат %8s. Під час третього запуску в **name2** потрапляють останні букви з імені **Максиміліан**, оскільки другий ввід поновлюється там, де закінчився перший ввід: в даному випадку – всередині слова **Максиміліан**.

В залежності від природи бажаного вводу, для читання тексту з клавіатури може бути зручніше користуватися функцією **fgets()**. Наприклад, функція **scanf()** не дуже підходить для вводу назви книги або пісні, якщо тільки назва не складається з одного слова. Типовим застосуванням **scanf()** є читання і перетворення даних різного типу в

деякій стандартній формі. Наприклад, якщо вхідний рядок містить назву інструмента, складський номер і ціну за штуку, то можна скористатися **scanf()** або написати власну функцію, яка буде виконувати перевірку на предмет помилок вводу. Якщо ви бажаєте обробляти ввід по слову за раз, можете застосовувати **scanf()**.

Функція **scanf()** страждає тим же потенційним недоліком, що й **gets()**: вона може призводити до переповнення, якщо слово, яке вводиться, не вміщується в цільову область пам'яті. Але для запобігання такого переповнення можна використовувати параметр ширини поля в специфікаторі %s.

7.1.6. Функції для виводу рядків

Тепер давайте перейдемо від вводу рядків до їх виводу. Ми знову будемо застосовувати бібліотечні функції. Для виводу рядків в **C** доступні три стандартні бібліотечні функції: **puts()**, **fputs()** і **printf()**.

Функція **puts()** дуже проста у використанні. Їй треба тільки передати в якості аргументу адресу рядка. В наступній програмі демонструється декілька способів з множини доступних.

```
#include <stdio.h>
#include <windows.h>
#define DEF "Я - рядок, який визначений директивою #define."

int main(void)
{
    char str1[80] =
        "Масив був ініціалізований певним значенням.";
        const char * str2 =
        "Вказівник був ініціалізований певним значенням.";

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    puts("Я - аргумент функції puts().");
    puts(DEF);
    puts(str1);
    puts(str2);
    puts(&str1[5]);
    puts(str2+4);
```

```
    return 0;
}
```

Результат роботи програми наведено на рис. 7.15.

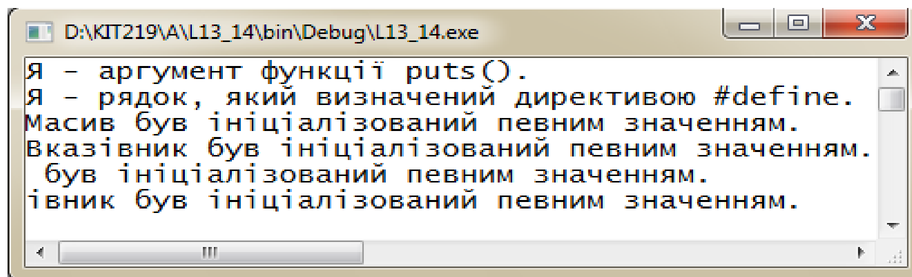


Рисунок 7.15 – Використання функції **puts()**

Як і в попередніх програмах, кожен рядок виводиться в окремому рядку, тому що при відображенні рядка функція **puts()** автоматично додає символ нового рядка.

Цей приклад нагадує, що фрази, які позначені подвійними лапками, являють собою рядкові константи і трактуються як адреси. Крім того, імена рядкових масивів також вважаються адресами. Вираз **&str1[5]** – це адреса шостого елемента в масиві **str1**. Цей елемент містить символ **'␣'**, і саме він **puts()** використовує в якості початкової точки. Аналогічно, **str2 + 4** вказує на комірку пам'яті, яка містить другий символ **'i'** з рядка "Вказівник", тому вивід починається з нього.

Як функція **puts()** дізнається, коли зупинитися? Вона припиняє вивід, коли зустрічає нульовий символ, так що краще, щоб він був у рядку. Не повторюйте помилку, яка розглядається в наступній програмі.

```
#include <stdio.h>
#include <windows.h>
int main(void)
{
    char side_a[] = "Сторона A";
    char dont[] = { 'У', 'Р', 'А', '!' };
    char side_b[] = "Сторона Б";

    SetConsoleOutputCP(1251);
    puts(dont);            // dont не є рядком
    return 0;
}
```

Оскільки в масиві **dont** відсутній завершальний нульовий символ, він не є рядком, тому функція **puts()** не знає, де зупинитися. Вона буде виводити вміст комірок пам'яті, що йдуть за **dont**, поки не виявить нульовий символ десь в іншому місці. Щоб гарантувати, що нульовий символ не виявиться надто далеко, масив **dont** в програмі зберігається між двома справжніми рядками. Результат роботи програми представлений на рис. 7.16.

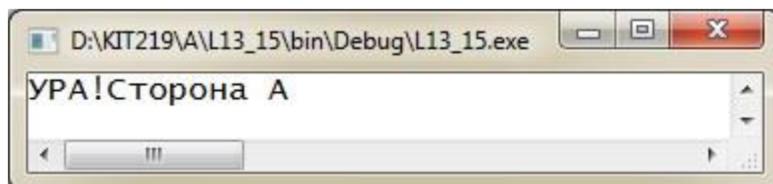


Рисунок 7.16 – Особливості роботи функції **puts()**

Функція **fputs()** являє собою версію **puts()**, що орієнтована на файли. Важливі відмінності між ними полягають у наступному:

1) Функція **fputs()** приймає другий аргумент, що вказує файл, в який повинен відбуватися запис. Для виводу на консоль можна застосовувати аргумент **stdout** (від **standard output** – стандартний вивід), який визначений в **stdio.h**.

2) На відміну від **puts()**, функція **fputs()** не додає автоматично до виводу символ нового рядка.

Зверніть увагу, що **gets()** відкидає символ нового рядка з введених даних, але **puts()** додає його до виводу. З іншого боку, **fgets()** зберігає символ нового рядка в даних, що введені, а **fputs()** не поміщає його в вивід.

Припустимо, що ви бажаєте реалізувати цикл, в якому рядок читається та виводиться в наступному рядку на консолі. В цьому випадку можна зробити наступним чином:

```
char line[81];
while(gets(line)) // еквівалентно while(gets(line) !=
    NULL) puts(line);
```

Згадайте, що функція **gets()** повертає нульовий вказівник, якщо

виявляє кінець файлу. Нульовий вказівник інтерпретується як нуль, або хибне значення, тому цикл припиняється. Можна зробити також наступним чином:

```
char line[81];
while(fgets(line, 81,
stdin))
   fputs(line, stdout);
```

В першому циклі рядок з масиву `line` відображається у власному рядку на екрані, оскільки `puts()` додає символ нового рядка. В другому циклі рядок з масиву `line` відображається у власному рядку на екрані через те, що `fgets()` зберігає символ нового рядка.

Слід відзначити, що якщо ви змішуєте ввід `fgets()` з виводом `puts()`, то отримаєте по два символи нового рядка для кожного рядка на екрані. Важливо розуміти, що функція `puts()` спроектована для роботи з `gets()`, а функція `fputs()` – для роботи з `fgets()`.

Функція `printf()` є менш зручною у використанні, ніж `puts()`, але вона є більш універсальною, оскільки здатна форматувати різні типи даних.

На відміну від `puts()`, функція `printf()` не виводить автоматично кожний рядок з нового рядка на екрані. Замість цього необхідно самостійно вказувати, де повинні починатися нові рядки. Таким чином,

```
printf("%s\n", string);
```

призводить до того ж самого результату, що й

```
puts(string);
```

Перша форма довше і потребую більше часу на виконання (правда, не настільки, щоб це стало помітно). З іншого боку, `printf()` спрощує об’єднання декількох рядків в одному рядку виводу. Наприклад, наступний оператор об’єднує в один рядок виводу слово `"Добре"`, ім’я користувача та символьний рядок, що визначається за допомогою `#define`:

```
printf("Добре, %s, %s\n", name, MSG);
```

7.1.7. Можливість самостійного створення функцій вводу-виводу

При вводі та виводі ви не обмежені тільки функціями стандартної бібліотеки **C**. Якщо вони недоступні або чомусь вам не подобаються, можете підготувати власні версії на основі функцій **getchar()** і **putchar()**. Припустимо, що вам потрібна функція, подібна до функції **puts()**, яка не додає автоматично символ нового рядка. В наступній програмі продемонстровано один зі способів створення такої функції.

```
#include <stdio.h>

void put1(const char * string)     // рядок не змінюється
{
    while(*string != '\0')
        putchar(*string++);
}
```

Вказівник **string** на тип **char** спочатку посилається на перший елемент аргументу, що передається. Оскільки ця функція не змінює рядок, застосовується модифікатор **const**. Після того, як вміст цього елемента виведено, вказівник інкрементується і вказує на наступний елемент. Це продовжується до тих пір, поки вказівник не буде посилатися на елемент, що містить нульовий символ. Згадайте, що операція **++** має більш високий пріоритет, ніж *****, тому виклик **putchar(*string++)** виводить значення, на яке вказує **string**, але інкрементує сам вказівник **string**, а не символ, на який він посилається.

Функцію **put1()** можна розглядати як модель для написання функцій обробки рядків. Оскільки кожний рядок містить нульовий символ, який позначає її кінець, передавати функції розмір рядка не треба. Замість цього функція обробляє символи по черзі, поки не зустріне нульовий символ.

Дещо більш довгий код функції передбачає використання форми запису з масивом:

```
int i = 0;
while(string[i] != '\0')
    putchar(string[i++]);
```

В даному випадку необхідна додаткова змінна для індексу.

Багато хто з програмістів на **C** будуть застосовувати наступну перевірку для циклу **while**:

```
while(*string)
```

Коли **string** вказує на нульовий символ, ***string** має значення **0**, що припиняє цикл. Такий підхід певним чином потребує меншого набору з клавіатури, ніж попередня версія. Тим, хто не знайомий з практикою програмування на **C**, цей прийом менш очевидний. Однак даний підхід дуже поширений, і очікується, що програмісти на **C** повинні його знати.

Припустимо, що необхідна функція, схожа на **puts()**,яка також повідомляє, скільки символів було виведено на консоль. Текст такої функції буде мати такий вигляд:

```
#include <stdio.h>

int put2(const char * string)
{
    int count = 0;
    while(*string)     // загальноприйнятий підхід {
        putchar(*string++);
        count++;
    }
    putchar('\n');    // символ нового рядка не враховується
    return(count);
}
```

Наступний виклик функції виводить рядок `"вітер"`:

```
put1("вітер");
```

Зазначений нижче виклик повертає також кількість символів, що присвоюються змінній **num** (в даному випадку **5**):

```
num = put2("вітер");
```

В наступній програмі наведено драйвер, що призначений для тестування **put1()** і **put2()**, а також вкладених викликів цих функцій:

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

```
  void put1(const char *);
  int put2(const char *);

  int main(void)
  {
      SetConsoleOutputCP(1251);

      put1("Програмісти левову частку часу проводять,\n");
      put1("читаючи код і розбираючись у ньому,\n");
      printf("На консоль виведено %d символів.\n",
             put2("тому важливо робити його зрозумілим\n" "як
             для себе, так і для колег."));
      return 0;
      }
void put1(const char * string)
{
    while(*string)        // еквівалентно *string != '\0'
    putchar(*string++);
}

int put2(const char * string)
{
    int count = 0;

    while(*string)
    {
        putchar(*string++);
        count++;
    }
    putchar('\n');
    return(count);
}
```

Результат роботи програми наведено на рис. 7.17.

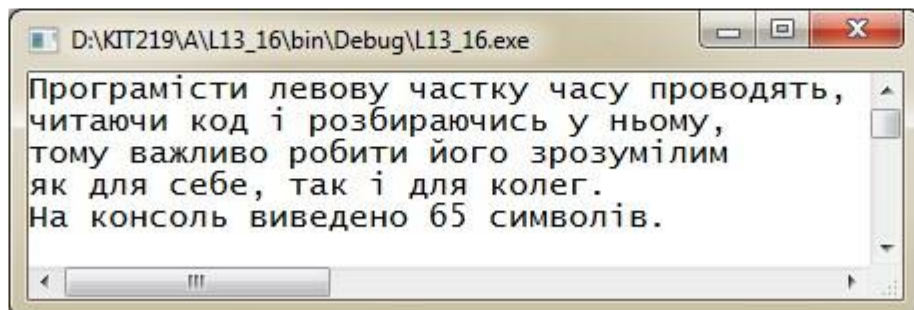


Рисунок 7.17 – Тестування функцій `put1()` і `put2()`

7.2. Функції для роботи з рядками

Бібліотека C надає декілька функцій для роботи з рядками. В ANSI C прототипи цих функцій містяться у файлі заголовку string.h. Ми з вами познайомимося з найбільш корисними та поширеними функціями: strlen(), strcat(), strncat(), strcmp(), strncmp(), strcpy() и strncpy(). Ми також дослідимо функцію sprintf(), яка підтримується файлом заголовку stdio.h.

7.2.1. Функція strlen()

Прототип функції:

```
size_t strlen(const char * string);
```

Тип size_t визначається відносно арифметичних можливостей процесора в файлі <stddef.h>. size_t – це беззнаковий цілий тип, що призначений для представлення розміру будь-якого об'єкта в пам'яті для конкретної реалізації. Оператор sizeof повертає значення типу size_t. Максимальний розмір size_t записаний в макроконстанті SIZE_MAX, яка визначена у файлі заголовку <stdint.h>. size_t повинен бути, як мінімум, 16 біт.

Функція strlen() визначає довжину рядка і виконується поки не зустріне нульовий символ, який не входить до загальної довжини рядка. Вона використовується в наступному прикладі функції, яка скорочує довгі рядки:

```
void fit(char *string, unsigned int size)
{
    if(strlen(string) > size) string[size] = '\0';
}
```

Функція змінює рядок, тому в її заголовку при оголошенні формального

параметра **string** модифікатор **const** не вказаний.

Функцію **fit()** можна протестувати за допомогою наступної програми. Зверніть увагу, що в коді застосовується конкатенація рядкових літералів **C**. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <string.h> // містить прототипи рядкових функцій

void fit(char *, unsigned int);
int main(void)
{
    char mesg[] = "Все повинно бути максимально
                  простим," " але не більше.";
    puts(mesg);
    fit(mesg, 35);
    puts(mesg);
    puts("Розглянемо ще декілька рядків.");
    puts(mesg + 36);
    return 0;
}
void fit(char *string, unsigned int size)
{
    if(strlen(string) > size) string[size] = '\0';
}
```

Результат виконання програми наведено на рис. 7.18.

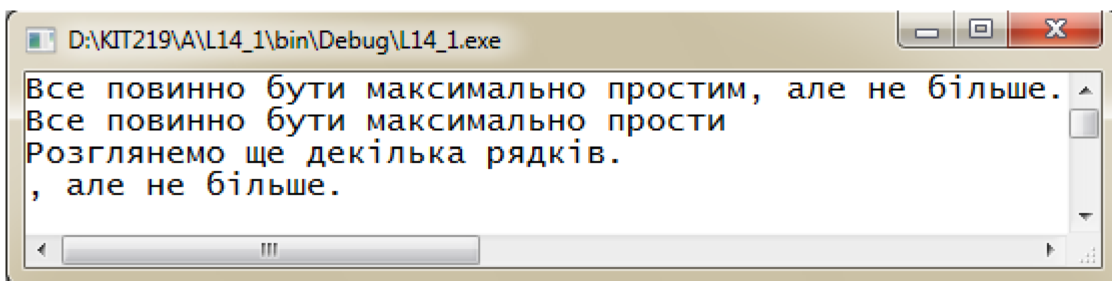


Рисунок 7.18 – Програма для скорочення довжини рядка, що виводиться

Функція **fit()** поміщає символ **'\0'** в **36**-й елемент масиву замість символу **'м'**. Вона зупиняється при виявленні першого нульового символу, ігноруючи частину масиву, що залишилася. Проте, частина масиву, що залишилася, нікуди не поділася, як про це свідчить наступний виклик функції:

```
puts(mesg + 36);
```

Вираз **mesg + 36** дає адресу елемента **mesg[36]**, яким є символ **','**. Таким чином, **puts()** відображає цей символ і продовжує роботу до тих пір,

поки не зіштовхнеться з початковим нульовим символом.

7.2.2. Функція strcat()

Прототип функції

```
char * strcat(char * destptr, const char * srcptr);
```

Функція **strcat()** (от **string concatenation** – конкатенація рядків) в якості аргументів приймає два рядки. Вона додає копію рядка **srcptr** наприкінці рядка **destptr**. Нульовий символ кінця рядка **destptr** замінюється першим символом рядка **srcptr**, і новий нульовий символ додається в кінець вже нового рядка, що сформований об'єднанням символів двох рядків в рядку **destptr**. Функція **strcat()** повертає тип **char *** (тобто вказівник на **char**) – адресу першого символу рядка, наприкінці якого був доданий другий рядок.

В наступній програмі демонструються можливості функції **strcat()**. В коді також застосовується функція **s_gets()**, яка була визначена на минулій лекції. Згадайте, що **s_gets()** використовує **fgets()** для читання рядка, а потім видаляє з нього символ нового рядка, якщо він є.

Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <string.h>      // оголошення функції strcat()
#define SIZE 80

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

int main(void)
{
    char flower[SIZE];
    char addon[] = " пахне як старі валянки.";

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    puts("Яка ваша улюблена квітка?");
    if(s_gets(flower, SIZE))
    {
        strcat(flower, addon);
        puts(flower);
        puts(addon);
```

```
    }
    else
        puts("Знайдено кінець файлу!");
        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)
    {
    while(st[i] != '\n' && st[i] != '\0')
        i++;

        if(st[i] == '\n')
           st[i] = '\0';
        else
           while(getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

Результат виконання програми наведено на рис. 7.19.

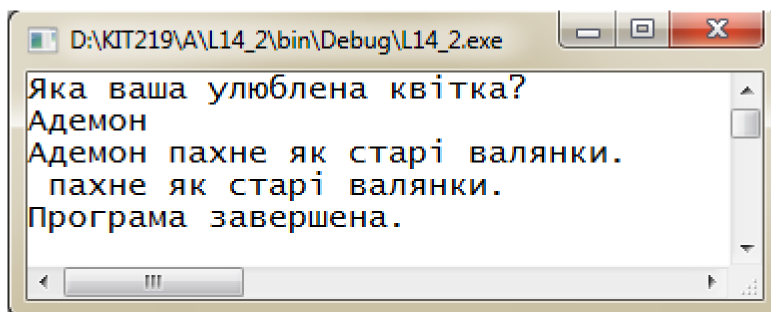


Рисунок 7.19 – Конкатенація двох рядків за допомогою функції **strcat()**

У виводі видно, що рядок **flower** змінився, а рядок **addon** – ні.

7.2.3. Функція strncat( )

Прототип функції

```
char * strncat(char * destptr, char * srcptr, size_t num);
```

Функція додає перші **num** символів рядка **srcptr** наприкінці рядка

destptr, плюс символ кінця рядка. Якщо рядок **srcptr** більше ніж кількість символів **num**, що копіюються, то після скопійованих символів неявно додається символ кінця рядка.

Функція **strcat()** не перевіряє, чи вміщується другий рядок у перший масив. Якщо ви не надасте достатнього простору для першого масиву, то матимете проблему, що пов'язана з переповненням сусідніх комірок пам'яті надлишковими символами.

Безумовно, можна заздалегідь перевірити довжину за допомогою **strlen()**.

Зверніть увагу, що ця функція додає **1** до загальної довжини, резервуючи місце для нульового символу. В якості альтернативи можна скористатися функцією **strncat()**, яка приймає другий аргумент, що вказує максимальну кількість символів, які додаються. Наприклад, виклик **strncat(**bugs, addon, 13**)** додає вміст рядка **addon** до **bugs**, зупиняючись після проходу **13** додаткових символів або при виявленні нульового символу, в залежності від того, що відбудеться раніше. Отже, враховуючи нульовий символ (який додається у будь-якому випадку), масив **bugs** повинен мати розмір, достатній для зберігання початкового рядка (без нульового символу), максимум **13** додаткових символів і завершального нульового символу. В наступній програмі ця інформація застосовується для обчислення значення змінної **available**, яка є максимальною дозволеною кількістю додаткових символів.

```
#include <stdio.h>
#include <windows.h>
#include <string.h>
#define SIZE 30
#define BUGSIZE 13

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

int main(void)
{
    char flower[SIZE];
    char addon[] = " пахне як старі валянки.";
```

```
    char bug[BUGSIZE];
    int available;

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    puts("Яка ваша улюблена квітка?");
    s_gets(flower, SIZE);
    if((strlen(addon) + strlen(flower) + 1) <= SIZE)
        strcat(flower, addon);
    puts(flower);
    puts("Яка ваша улюблена комаха?");
    s_gets(bug, BUGSIZE);
    available = BUGSIZE - strlen(bug) - 1;
    strncat(bug, addon, available);
    puts(bug);
    return 0;
}
char *s_gets(char *st, int n)
{
    char *ret_val;
    int i = 0;

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

Результат виконання програми наведено на рис. 7.20.

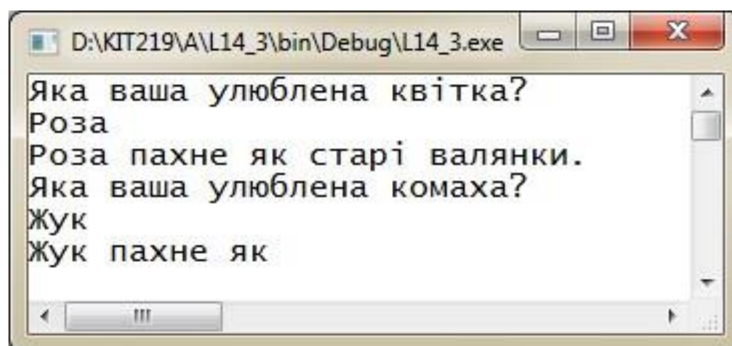


Рисунок 7.20 – Конкатенація двох рядків за допомогою функції **strncat()**

Ви вже могли помітити, що функція **strcat()**, як і **gets()**, може призводити до переповнення буфера. Чому ж тоді в стандарті **C11** не відмовилися від **strcat()**, а лише запропонували функцію **strncat()**? Однією з причин може бути те, що функція **gets()** піддає програму небезпеці з боку тих, хто її використовує, в той час як **strcat()** піддає програму небезпеці внаслідок неуважності програміста. Майбутню поведінку того чи іншого користувача передбачити неможливо, але можна контролювати те, що відбувається всередині програми. Філософія довіри програмісту, що була прийнята в **C**, покладає на нього відповідальність за визначення ситуацій, в яких функція **strcat()** може застосовуватися безпечним чином.

7.2.4. Функція strcmp()

Прототип функції:

```
int strcmp(const char *string1, const char *string2);
```

Функція **strcmp()** по черзі порівнює символи двох рядків **string1** і **string2** поки не буде досягнутий кінець рядка. Як тільки будуть знайдені перші неоднакові символи, функція проаналізує числові коди цих символів. Чий код виявиться більшим, той рядок і буде вважатися більшим.

Функція повертає декілька значень: нульове значення говорить про те, що обидва рядки збігаються; значення більше нуля вказує на те, що рядок **string1** більше ніж рядок **string2**; значення менше нуля свідчить про зворотне.

Припустимо, що необхідно порівняти введену кимось відповідь з рядком, що зберігається в пам’яті, як показано в наступній програмі:

```
#include <stdio.h>
#include <windows.h>
#define ANSWER "Грант"
#define SIZE 40

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

```
int main(void)
{
	char mas[SIZE];

	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	puts("Хто похований в могилі Гранта?");
	s_gets(mas, SIZE);
	while(strcmp(mas, ANSWER) != 0)
	{
		puts("Неправильно! Спробуйте ще раз.");
		s_gets(mas, SIZE);
	}
	puts("Тепер правильно!");
	return 0;
}
char *s_gets(char *st, int n)
{
	char *ret_val;
	int i = 0;

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

Результат виконання програми наведено на рис. 7.21 і рис. 7.22.

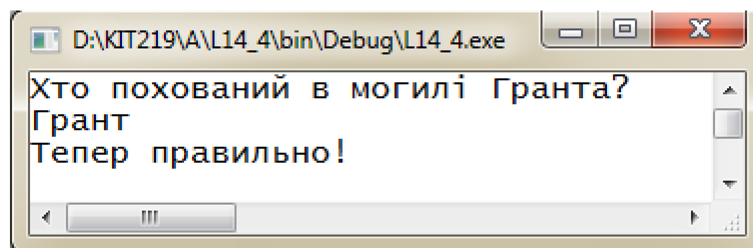


Рисунок 7.21 – Порівняння двох рядків за допомогою функції **strcmp()**: результат порівняння хибний

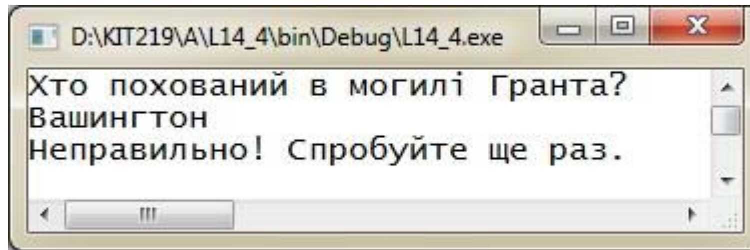


Рисунок 7.22 – Порівняння двох рядків за допомогою функції **strcmp()**: результат порівняння успішний

Одна з примітних властивостей функції **strcmp()** полягає в тому, що вона порівнює рядки, а не масиви. Хоча масив **mas** займає **40** комірок пам'яті, а рядок **"Грант"** – тільки шість (одна комірка відводиться під нульовий символ), при порівнянні переглядається тільки частина масиву **mas** до першого нульового символу. Отже, **strcmp()** може використовуватися для порівняння рядків, що зберігаються в масивах різних розмірів.

Що буде, якщо користувач введе в якості відповіді **"ГРАНТ"**, **"грант"** або **"Улісс С. Грант"**? Програма повідомить про неправильну відповідь. Щоб зробити програму більш дружньою, необхідно передбачити всі можливі правильні відповіді. Для цього є декілька прийомів. Наприклад, за допомогою директиви **#define** можна визначити відповідь як **"ГРАНТ"** і написати функцію, яка перетворює усі введені літери на прописні. Це усуває проблему використання прописних літер, але як і раніше залишаються інші форми відповіді, про які слід потурбуватися, а також врахувати той факт, що дружина Гранта, Джулія, також похована в цій могилі.

Розглянемо програму, яка працює з різними варіантами використання функції **strcmp()**. Текст програми має такий вигляд:

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

int main(void)
{
    SetConsoleOutputCP(1251);
    printf("strcmp(\"A\", \"A\") повертає ");
    printf("%2d\n", strcmp("A", "A"));

    printf("strcmp(\"A\", \"B\") повертає ");
    printf("%2d\n", strcmp("A", "B"));

    printf("strcmp(\"B\", \"A\") повертає ");
    printf("%2d\n", strcmp("B", "A"));

    printf("strcmp(\"C\", \"A\") повертає ");
    printf("%2d\n", strcmp("C", "A"));

    printf("strcmp(\"Z\", \"a\") повертає ");
    printf("%2d\n", strcmp("Z", "a"));

    printf("strcmp(\"apples\", \"apple\") повертає ");
    printf("%2d\n", strcmp("apples", "apple"));

    printf("strcmp(\"вікно\", \"вікно\")  повертає ");
    printf("%2d\n", strcmp("вікно", "вікно"));

    printf("strcmp(\"вікно\", \"вітер\")  повертає ");
    printf("%2d\n", strcmp("вікно", "вітер"));

    printf("strcmp(\"річка\", \"ліс\")    повертає ");
    printf("%2d\n", strcmp("річка", "ліс"));
    return 0;
}
```

Результат виконання програми наведено на рис. 7.23.

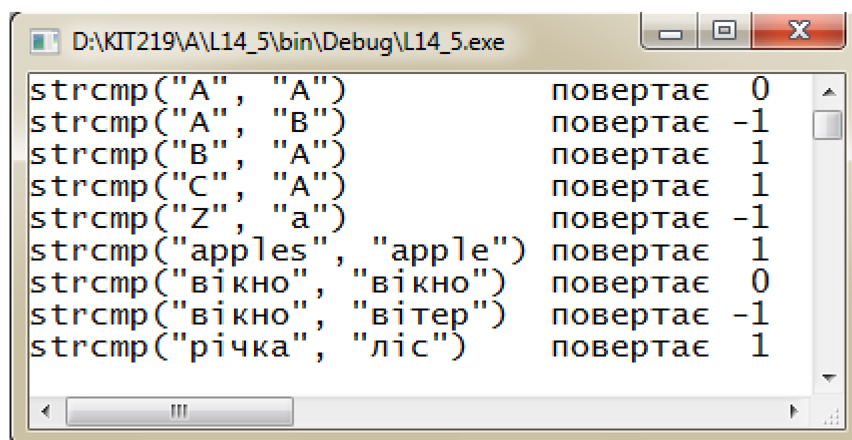


Рисунок 7.23 – Різні варіанти порівняння двох рядків за допомогою функції **strcmp()**

Наприклад, якщо необхідно знайти рядки, що починаються з **"астро"**, то пошук можна було б обмежити першими п’ятьма символами. Текст програми, що здійснює такий пошук має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <string.h>
#define LISTSIZE 6

int main(void)
{
    const char *list[LISTSIZE] = {
                                 "астрономія", "астеризм",
                                 "астрофізика","остракізм",
                                 "астатизм", "астролябія"
                                };
    int count = 0;
    SetConsoleOutputCP(1251);
    for(int i = 0; i < LISTSIZE; i++)
    {
        if(strncmp(list[i], "астро", 5) == 0)
        {
            printf("Знайдено: %s\n", list[i]);
            count++;
        }
    }
    printf("Кількість слів у списку,"
           " що починаються на астро: %d\n", count);
    return 0;
}
```

Результат виконання програми наведено на рис. 7.24.

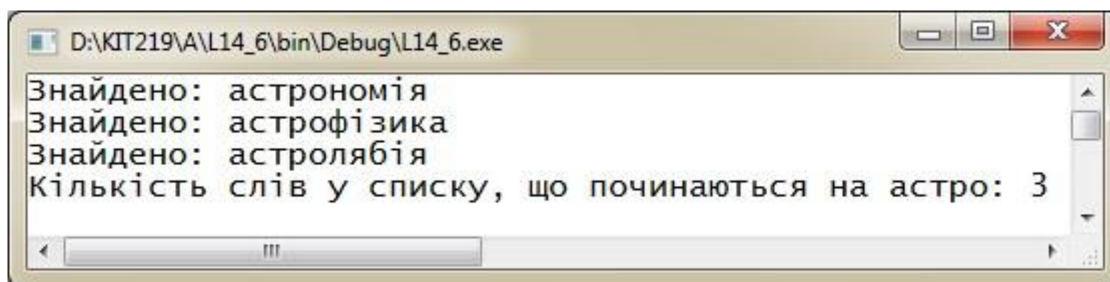


Рисунок 7.24 – Пошук рядків за допомогою функції **strcmp()**, які починаються з фрагмента "астро"

7.2.5. Функції strcpy()і strncpy()

Прототипи функцій:

```
char *strcpy(char *destptr, const char *srcptr);
char *strncpy(char *destptr, const char *srcptr, size_t num);
```

Функція **strncpy()** копіює перші **num** символів рядка **srcptr** до рядка **destptr**. Якщо кінець рядка **srcptr** (символ кінця рядка) досягнутий перш ніж були скопійовані **num** символів, до скопійованих символів наприкінці рядка **destptr** додається нульовий символ, після чого, рядок вважається скопійованим. Якщо ж рядок призначення виявиться меншим за **num**, тоді будуть скопійовані символи, які помістяться в **destptr**, враховуючи, що наприкінці рядка обов'язково повинен бути символ кінця рядка.

Раніше було зазначено, що якщо **pts1** і **pts2** – вказівники на рядки, то оператор **pts2 = pts1;** копіює тільки адресу рядка, а не сам рядок. Проте, припустимо, що необхідно скопіювати рядок. В такому випадку можна скористатися функцією **strcpy()**. Код в наступній програмі пропонує користувачу ввести слова, що починаються з букви **'к'**. Ця програма копіює ввід у тимчасовий масив, і, якщо першою літерою є **'к'** програма використовує функцію **strcpy()** для копіювання цього рядка з тимчасового масиву на місце її постійного зберігання. Функція **strcpy()** являє собою рядковий еквівалент оператора присвоювання.

```
#include <stdio.h>
#include <windows.h>
#include <string.h>        // оголошення strcpy()
#define SIZE 40
#define LIM 5

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

int main(void)
{
    char qwords[LIM][SIZE];
    char temp[SIZE];
    int i = 0;

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    printf("Введіть %d слів, які починаються"
           " з літери \'к\':\n", LIM);
    while(i < LIM && s_gets(temp, SIZE))
```

```
    {
        if(temp[0] != 'к')
            printf("%s не починається з літери \'к\'!\n", temp);
        else
        {
            strcpy(qwords[i], temp);
            i++;
        }
    }
    puts("Список правильних слів:");
    for(i = 0; i < LIM; i++)
        puts(qwords[i]);
    return 0;
}

char *s_gets(char *st, int n) {
    char *ret_val;
    int i = 0;

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

Результат виконання програми наведено на рис. 7.25.

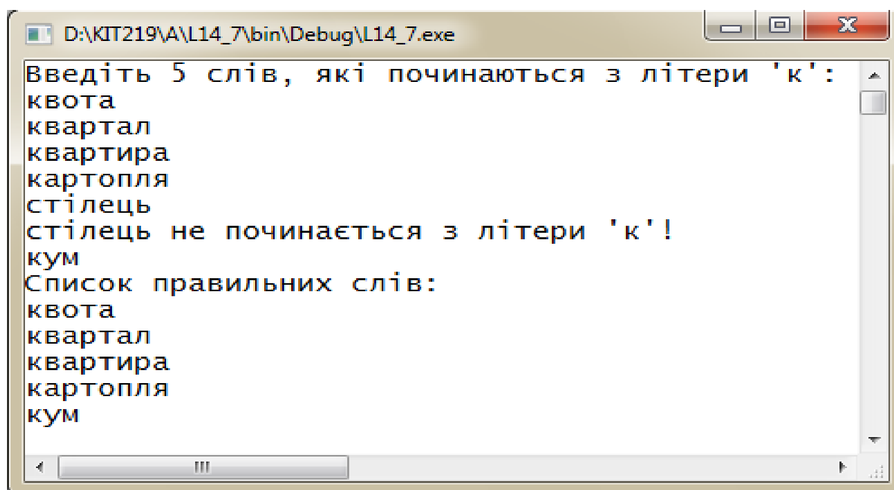


Рисунок 7.25 – Приклад використання функції **strcpy()** для виводу на консоль слів, які починаються з літери **'к'**

Функція **strcpy()** має ще дві властивості, які можна вважати зручними. По-перше, її типом є **char** *. Вона повертає значення свого першого аргументу – адреси символу. По-друге, перший аргумент не обов’язково повинен вказувати на початок масиву. Це дозволяє копіювати тільки частину масиву. Розглянемо програму, де використовуються обидві ці властивості. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <string.h>     // оголошення strcpy()
#define WORDS "найгіршим"
#define SIZE 40

int main(void)
{
    const char *orig = WORDS;
    char copy[SIZE] = "Будьте кращим, ніж могли б бути.";
    char *ps;

    SetConsoleOutputCP(1251);

    puts(orig);
    puts(copy);
    ps = strcpy(copy + 7, orig);
    puts(copy);
    puts(ps);
    return 0;
}
```

Результат роботи програми наведено на рис. 7.26.

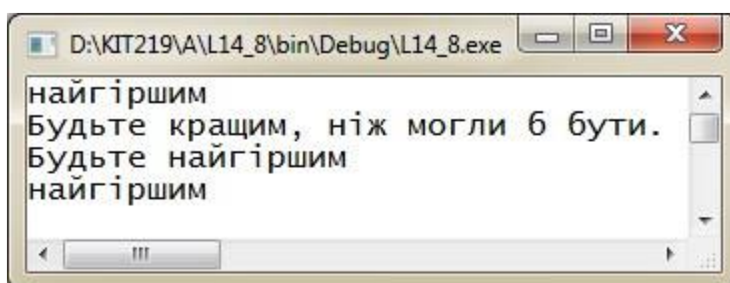


Рисунок 7.26 – Приклад використання функції **strcpy()** для копіювання частини символьного масиву

Функція **strcpy()** має ту ж саму проблему, що й **strcat()** – жодна з них не перевіряє, чи вміщується початковий рядок в цільовий рядок. Більш безпечний спосіб копіювання рядків передбачає застосування функції

strncpy(). Ця функція приймає третій аргумент, в якому вказується максимальна кількість символів, що копіюються. В наступній програмі замість **strcpy()** використовується **strncpy()**. Щоб показати, що відбувається у випадку, коли розмір початкового рядка занадто великий для цільових рядків, в коді обраний невеликий розмір (сім елементів, шість символів).

```
#include <stdio.h>
#include <windows.h>
#include <string.h>     // оголошення strncpy()
#define SIZE 40
#define TARGSIZE 7
#define LIM 5

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

int main(void)
{
    char qwords[LIM][TARGSIZE];
    char temp[SIZE];
    int i = 0;

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    printf("Введіть %d слів, які починаються"
           " з літери \'к\':\n", LIM);
    while(i < LIM && s_gets(temp, SIZE))
    {
        if(temp[0] != 'к')
            printf("%s не починається "
                   " з літери \'к\'!\n", temp);
        else
        {
            strncpy(qwords[i], temp, TARGSIZE - 1);
            qwords[i][TARGSIZE - 1] = '\0';
            i++;
        }
    }
    puts("Список правильних слів:");
    for(i = 0; i < LIM; i++)
        puts(qwords[i]);
    return 0;
}

char *s_gets(char *st, int n)
{
    char *ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
```

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

Результат роботи програми наведено на рис. 7.27.

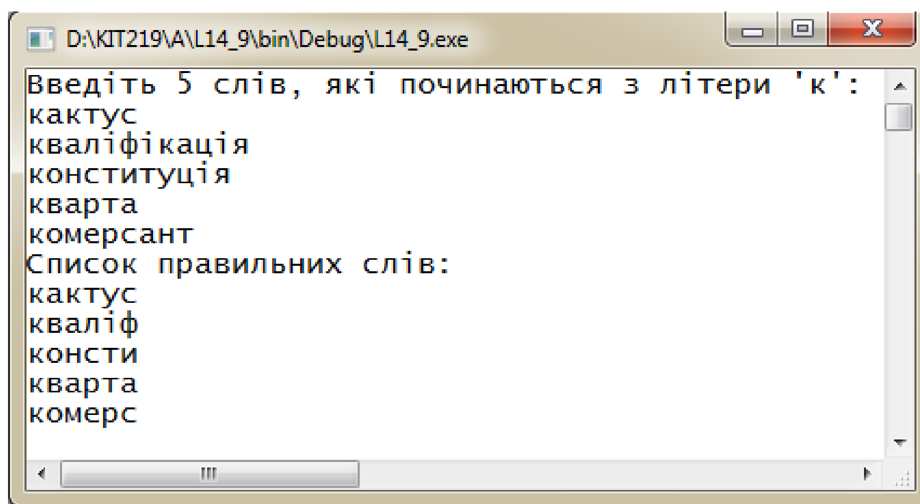


Рисунок 7.27 – Приклад використання функції **strncpy()** для копіювання визначеної кількості літер

Виклик функції **strncpy(**target, source, n**)** копіює аж до **n** символів або до появи нульового символу (в залежності від того, що відбудеться раніше) з **source** в **target**. Отже, якщо кількість символів в **source** менше **r**, копіюється увесь рядок, включаючи нульовий символ. Ця функція ніколи не копіює більш ніж **n** символів, тому якщо даний ліміт вичерпаний до досягнення кінця початкового рядка, то нульовий символ не додається. Таким чином, фінальний результат може містити, а може й не містити нульовий символ. З цієї ж причини значення **n** в програмі обрано на одиницю менше розміру цільового масиву, а останній елемент масиву

встановлений в нульовий символ:

```
strncpy(qwords[i], temp, TARGSIZE - 1);
qwords[i][TARGSIZE - 1] = '\0';
```

Це забезпечує зберігання рядка. Якщо початковий рядок насправді вміщується в цільовий рядок, то скопійований разом з ним нульовий символ помічає справжній кінець рядка. Якщо початковий рядок в цільовий не вміщується, то кінець рядка помічається останнім нульовим символом.

7.2.6. Функція sprintf()

Прототип функції:

```
int sprintf(char *buffer, const char *format, ...);
```

Функція `sprintf()` оголошена в файлі заголовку `stdio.h`, а не в `string.h`. Вона працює подібно `printf()`, але здійснює запис в рядок, а не на екран. Таким чином, вона надає спосіб об'єднання декількох елементів в єдиний рядок. Перший аргумент `sprintf()` – це адреса цільового рядка. Решта аргументів аналогічні аргументам в `printf()` – рядок специфікації перетворення і список елементів, призначених для запису.

В наступній програмі функція `sprintf()` застосовується для об'єднання трьох елементів (двох рядків і числа) в один рядок. Зверніть увагу, що `sprintf()` використовується так само, як це б робилося у випадку функції `printf()`, крім того, що вихідний рядок зберігається в масиві `formal`, а не відображається на екрані. Текст програми має такий вигляд:

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

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

int main(void)
{
    char first[MAX];
    char last[MAX];
    char formal[2 * MAX + 10];
    double prize;
    SetConsoleCP(1251);
```

```
    SetConsoleOutputCP(1251);
    puts("Введіть своє ім\'я:");
    s_gets(first, MAX);
    puts("Введіть своє прізвище:");
    s_gets(last, MAX);
    puts("Введіть суму грошового призу:");
    scanf("%lf", &prize);
    sprintf(formal, "%s, %-19s: $%6.2f\n", last, first,prize);
    puts(formal);
    return 0;
}
char *s_gets(char *st, int n)
{
    char *ret_val;
    int i = 0;

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

Результат виконання програми наведено на рис. 7.28.

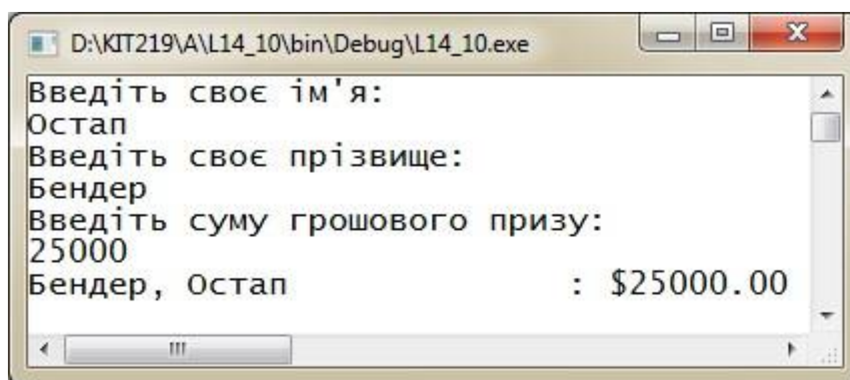


Рисунок 7.28 – Приклад використання функції **sprintf()** для об’єднання трьох елементів в один рядок

Функція **sprintf()** приймає вхідні дані і форматує їх в стандартному вигляді, після чого зберігає в рядку **formal**.

7.2.7. Інші функції для роботи з рядками

Бібліотека **ANSI C** містить більше **20** функцій, що призначені для роботи з рядками. Далі наведено короткий опис функцій, які найчастіше використовуються для роботи з рядками (більшість з них вже були розглянуті).

```
char *strcpy(char * restrict s1, const char * restrict s2);
```

Ця функція копіює рядок (включаючи нульовий символ), на який вказує **s2**, в комірку, на яку вказує **s1**. Значенням, що повертається, є **s1**.

Ключове слово **restrict** – в мові програмування **C** введено стандартом **C99** і використовується в оголошеннях вказівників. Воно дозволяє програмісту повідомити компілятор, що оголошений вказівник вказує на блок пам'яті, на який не вказує жоден інший вказівник. Гарантію того, що на один блок пам'яті не буде вказувати більш ніж один вказівник, дає програміст.

```
char *strncpy(char * restrict s1, const char * restrict s2,
              size_t n);
```

Ця функція копіює в комірку, на яку вказує **s1**, не більш ніж **n** символів з рядка, на який вказує **s2**. Значенням, що повертається, є **s1**. Символи, які йдуть за нульовим символом, не копіюються, і якщо початковий рядок коротше ніж **n** символів, частина цільового рядка, що залишилася, заповнюється нульовими символами. Якщо початковий рядок містить **n** або більшу кількість символів, нульовий символ не копіюється. Значенням, що повертається, є **s1**.

```
char *strcat(char * restrict s1, const char * restrict s2);
```

Рядок, на який вказує **s2**, копіюється в кінець рядка, на який вказує **s1**. Перший символ рядка **s2** копіюється поверх нульового символу рядка **s1**. Значенням, що повертається, є **s1**.

```
char *strncat(char *restrict s1, const char * restrict s2,
              size_t n);
```

До рядка **s1** додається не більше ніж **n** символів рядка **s2**, причому перший символ рядка **s2** копіюється поверх нульового символу рядка **s1**. Нульовий символ і будь-які інші символи, які за йдуть за ним в рядку **s2**, не копіюються, а до результату додається нульовий символ. Значенням, що повертається, є **s1**.

```
int strcmp(const char *s1, const char *s2);
```

Ця функція повертає додатне значення, якщо в послідовності зіставлення комп'ютера рядок **s1** йде за рядком **s2**, значення **0**, якщо рядки є ідентичними, і від'ємне значення, якщо в послідовності зіставлення перший рядок передує другому.

```
int strncmp(const char *s1, const char *s2, size_t n);
```

Ця функція працює подібно до **strcmp()**, за виключенням того, що процедура порівняння зупиняється після перегляду **n** символів або при появі першого нульового символу, в залежності від того, що відбудеться раніше.

```
char *strchr(const char *s, int c);
```

Ця функція повертає вказівник на першу комірку рядка **s**, в якому міститься символ **c**. Завершальний нульовий символ є частиною рядка, так що його теж можна шукати. Якщо символ не знайдено, функція повертає нульовий вказівник.

```
char *strpbrk(const char *s1, const char *s2);
```

Ця функція повертає вказівник на першу комірку рядка **s1**, в якому міститься будь-який символ, знайдений в рядку **s2**. Функція повертає нульовий вказівник, якщо жодного символу не знайдено.

```
char *strrchr(const char *s, int c);
```

Ця функція повертає вказівник на останнє входження символу **c** в рядку

s. Завершальний нульовий символ є частиною рядка, тому його також можна шукати. якщо символ не знайдено, функція повертає нульовий вказівник.

```
char *strstr(const char *s1, const char *s2);
```

Ця функція повертає вказівник на перше входження рядка s2 всередині рядка s1. Якщо рядок не знайдено, функція повертає нульовий вказівник.

```
size_t strlen(const char *s);
```

Ця функція повертає кількість символів, не включаючи нульовий, що знаходиться в рядку s.

Зверніть увагу, що в усіх прототипах застосовується ключове слово const, щоб позначити, які рядки не змінюються функцією. Наприклад, погляньте на такий прототип:

```
char *strcpy(char * restrict s1, const char * restrict s2);
```

Це означає, що s2 вказує на рядок, який не може бути змінений, у всякому разі, функцією strcpy(), але s1 вказує на рядок, змінювати який дозволено. В цьому є сенс, оскільки s1 – це цільовий рядок, який піддається змінам, а s2 – початковий рядок, який повинен залишатися незмінним.

Ключове слово restrict встановлює обмеження на те, як повинні застосовуватися аргументи функції, наприклад, воно вказує на неприпустимість копіювання рядка самого в себе.

Тип size_t – це будь-який тип, що повертається операцією sizeof. В мові C заявлено, що операція sizeof повертає цілочисловий тип, але не задано, який саме. Таким чином, в одній системі size_t може бути unsigned int, а в іншій – unsigned long. У файлі заголовку string.h тип має визначення size_t для конкретної системи або вказується посилання на інший файл заголовку, який містить необхідне визначення.

Розглянемо простий випадок використання однієї з таких функцій.

Раніше було показано, що функція **fgets()** при читанні рядка вводу зберігає символ нового рядка в цілому рядку. В нашій функції **s_gets()** для виявлення символу нового рядка використовується цикл **while**, але замість нього можна використовувати **strchr()**. Спочатку треба знайти за допомогою функції **strchr()** символ нового рядка, якщо він є. Коли його знайдено, **strchr()** повертає адресу символу нового рядка, і потім за цією адресою можна помістити нульовий символ:

```
fgets(line, 80, stdin);
find = strchr(line, '\n'); // пошук символу нового рядка
if(find)                    // якщо адреса не є NULL
   *find = '\0';            // помістити туди нульовий символ
```

Якщо **strchr()** не вдається знайти символ нового рядка, функція **fgets()** досягає ліміту на розмір ще до кінця рядка. Для обробки такої ситуації до оператора **if** можна додати конструкцію **else**, як це робилося в **s_gets()**.

7.2.8. Сортування рядків

Розглянемо практичну задачу сортування рядків в алфавітному порядку. Ця задача виникає під час підготовки списків прізвищ, під час індексації та в багатьох інших ситуаціях. Одним з основних інструментів в такій програмі є функція **strcmp()**, оскільки вона може застосовуватися для визначення порядку слідування двох рядків. Основними задачами, які буде вирішувати дана програма, є: читання масиву рядків, їх сортування та вивід. Буде також застосований один зі стандартних алгоритмів сортування. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <string.h>
#define SIZE 81
#define LIM 20
#define HALT ""

void stsrt(char *strings[], int num);
char *s_gets(char *st, int n);
```

```
  int main(void)
  {
    char input[LIM][SIZE]; // масив для вхідних даних
    char *ptstr[LIM];       // масив вказівників
    int ct = 0;             // лічильник вводу
    int k;                  // лічильник виводу

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    printf("Введіть до %d рядків, і вони"
           " будуть відсортовані.\n", LIM);
    printf("Щоб зупинити ввід, натисніть"
           " клавішу Enter на початку рядка\n");

    while(ct < LIM && s_gets(input[ct], SIZE) != NULL
          && input[ct][0] != '\0')
    {
        // встановлюємо вказівники на рядки
        ptstr[ct] = input[ct];
        ct++; }

    // сортувальник рядків
    stsrt(ptstr, ct);
    puts("\nВідсортований список:\n");
    for(k = 0; k < ct; k++)
        puts(ptstr[k]);    // відсортовані вказівники
    return 0;
}

// функція сортування вказівників на рядки
void stsrt(char *strings[], int num)
{
    char *temp;
    int top, seek;

    for(top = 0; top < num - 1; top++)
    {
        for(seek = top + 1; seek < num; seek++)
        {
            if(strcmp(strings[top], strings[seek]) > 0)
            {
                temp = strings[top];
                strings[top] = strings[seek];
                strings[seek] = temp;
            }
        }
    }
}

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

```
        int i = 0;
        ret_val = fgets(st, n, stdin);
          if(ret_val)
          {
              while(st[i] != '\n' && st[i] != '\0')
                  i++;
              if(st[i] == '\n')
                  st[i] = '\0';
              else
                 while(getchar() != '\n')
                      continue;
          }
          return ret_val;
    }
```

Результат виконання програми наведено на рис. 7.29.

```
D:\KIT219\A\L14_11\bin\Debug\L14_11.exe
Введіть до 20 рядків, і вони будуть відсортовані.
Щоб зупинити ввід, натисніть клавішу Enter на початку рядка
Мені тринадцятий минало.
Я пас ягнята за селом.
Чи то так сонечко сіяло,
Чи так мені чого було?


Відсортований список:

Мені тринадцятий минало.
Чи так мені чого було?
Чи то так сонечко сіяло,
Я пас ягнята за селом.
```

Рисунок 7.29 – Результат сортування рядків

Сортування вказівників замість рядків. Складна частина цієї програми пов'язана з тим, що замість самих рядків перевпорядковуються вказівники на ці рядки. Спочатку елемент **ptrst[0]** встановлюється в **input[0]** і т. д. В результаті вказівник **ptrst[i]** посилається на перший символ масиву **input[i]**. Кожен елемент **input[i]** являє собою масив з **81** елемента, а кожен елемент **ptrst[i]** – окрему змінну. Процедура сортування перевпорядковує **ptrst**, залишаючи **input** незмінним. Якщо, наприклад,

в алфавітному порядку **input[1]** знаходиться попереду **input[0]**, то програма переключить вказівники в **ptrst**, через що **ptrst[0]** буде посилатися на початок **input[1]**, а **ptrst[1]** – на початок **input[0]**. Це набагато простіше, ніж використання функції **strcpy()** для обміну вмістом двох рядків в **input**. Даний процес також має перевагу через те, що зберігає початковий порядок у масиві **input**.

Алгоритм сортування вибором. Для сортування вказівників був застосований алгоритм сортування вибором. Ідея полягає у використанні циклу **for** для порівняння усіх елементів по черзі з першим. Якщо елемент, що порівнюється, передує поточному першому елементу, вони міняються місцями. До моменту, коли досягається кінець циклу, перший елемент містить вказівник на рядок, що знаходиться першим в послідовності зіставлення комп'ютера. Потім зовнішній цикл **for** повторює процес, починаючи цього разу з другого елемента **input**. Коли внутрішній цикл завершиться, у другому елементі **ptrst** виявиться вказівник на рядок, що знаходиться другим в послідовності зіставлення. Процес продовжується до тих пір, поки не будуть відсортовані усі елементи.

Давайте більш докладно розглянемо сортування вибором. Нижче показаний алгоритм сортування на псевдокоді:

```
для n = перший елемент до n = передостанній елемент
    знайти найбільше з чисел, що залишилися,
    і помістити його в n-й елемент
```

Це працює наступним чином. Почніть з **n = 0**. Перегляньте увесь масив, знайдіть найбільше число та поміняйте його і перший елемент місцями. Далі встановіть **n** = 1 і перегляньте усі елементи масиву крім першого. Найдіть найбільший з елементів, що залишилися, і поміняйте його та другий елемент місцями. Продовжуйте цей процес до тих пір, поки не досягнете передостаннього елемента. Тепер залишилися тільки два елемента. Порівняйте їх і помістіть більший в позицію передостаннього елемента. У підсумку найменший елемент зайняв свою кінцеву позицію.

Виглядає так, що це задача для циклу **for**, але ми ще повинні більш докладно описати процес «знайти та помістити». Один зі способів вибору найбільшого значення з числа тих, що залишилися, передбачає порівняння першого та другого елементів в частині масиву, що залишилася. Якщо другий елемент більше першого, зробіть обмін їх значеннями. Далі порівняйте перший елемент з третім. Якщо третій елемент більше, обміняйте їх значеннями. Кожен обмін призводить до переміщення більшого елемента ближче до початку списку. Продовжуйте діяти подібним чином до тих пір, поки не відбудеться порівняння першого елемента з останнім. Після завершення найбільше значення виявиться в першому елементі масиву, що залишився. Отже, ви відсортували масив для першого елемента, однак інші елементи не відсортовані. Ось як можна представити процедуру за допомогою псевдокоду:

```
для n = передостанній елемент
    порівняти n-й елемент з першим елементом;
        якщо n-й елемент більше, обміняти їх значеннями
```

Цей процес виглядає як ще один цикл **for**. Він повинен бути вкладений в перший цикл **for**. Зовнішній цикл вказує, який елемент масиву повинен бути заповнений, а внутрішній цикл знаходить значення, яке в нього слід помістити. Об’єднавши разом обидві частини псевдокоду і представивши алгоритм на **C**, отримаємо нашу функцію. В бібліотеці **C** є більш досконала функція сортування **qsort()**. Вона приймає вказівник на функцію, що виконує порівняння під час сортування. Її робота буде продемонстрована в одній з подальших лекцій.

Контрольні запитання та завдання

1. З яких елементів масиву складається символьний рядок?
2. Перелічіть способи визначення рядків.
3. Як відбвається ініціалізація масивів символьних рядків?

4. В чому полягає різниця між формами у вигляді масиву та вказівника?

5. Поясніть різницю між ініціалізацією символьного масиву, який призначений для зберігання рядка, та ініціалізацією вказівника, який вказує на цей рядок. Що це означає на практиці?

6. Наведіть послідовність дій для створення вільного місця для рядка.

7. Чому ж тоді функції **gets()** приділяється особлива увага?

8. Які функції є *альтернативними функції* **gets()**?

9. Для чого призначена функція **fgets()**?

10. Що означає і для чого використовується буферизований ввід-вивід?

11. Нульовий вказівник, або **NULL,** яке має значення?

12. Різниця між нульовим символом та нульовим вказівником?

13. Наведіть основні відмінності функції **gets_s()** від **fgets()**.

14. В чому полягає основна відмінність між функціями **scanf()** і **fgets()**?

15. Перелічить стандартні бібліотечні функції для виводу рядків.

16. Продемоструйте способи використання функції **puts()**.

17. В чому полягає відмінність між функціями **fputs()** і **puts()**?

18. Наведіть можливісті, які існують для самостійного створення функцій вводу-виводу.

19. Які можливості надає бібліотека мови *C* для роботи з рядками?

20. Що визначають функції **strlen()** та **fit()**?

21. Для чого використовуються функції **strcat()** та **strncat().**

22. Які існують проблеми щодо викоистан функцій **strcpy()** і **strcat().**

23. Наведіть опис функцій, які найчастіше використовуються для роботи з рядками.

24. Які основні інструменти застосовуються для сортування рядків?

Завдання для самостійного розв'язання

1. Визначте трьома різними способами (за допомогою константного рядка, масиву типу `char` та вказівника на тип `char`) рядок символів, що складається з вашого прізвища, імені та по батькові, записаних через пробіл. Виведіть на консоль кожен з варіантів за допомогою функції `puts()`.

2. Напишіть *C*-програму, яка підраховує кількість голосних в рядку, які міістяться у вашому прізвищі, імені та по батькові, та виведіть на консоль номери їх позицій.

3. Напишіть *C*-програму, яка, наводить можливість застосування функцій `gets()` і `puts()`.

8. ФАЙЛОВИЙ ВВІД-ВИВІД

8.1. Функції для роботи з файлами

Файли є невід'ємною частиною сучасних комп'ютерних систем. Вони використовуються для збереження програм, документів, даних, кореспонденції, зображень, фотографій, музичних творів, відеокліпів і великої кількості інших видів інформації. Ви повинні вміти писати програми, які створюють файли, записують до них інформацію та зчитують її з файлів. В цій та наступній лекціях будуть розглянуті основні питання, які тим або іншим чином пов'язані з файлами.

8.1.1. Взаємодія з файлами

Мова **C** дозволяє відкривати файл всередині програми й потім за допомогою спеціальних функцій вводу-виводу здійснювати читання та запис даних. Перш ніж переходити до вивчення таких функцій, розглянемо коротко саму природу файлу.

Файл – це впорядкована сукупність даних, що зазвичай розташована на жорсткому диску або іншому зовнішньому носії інформації. Великий за розміром файл може зберігатися в декількох окремих фрагментах або містити додаткові дані, які дозволяють операційній системі визначати вид цього файлу.

В мові **C** файл розглядається як неперервна послідовність байтів, кожен з яких може бути прочитаний індивідуально.

Існують два режими представлення файлів: **текстовий** і **двійковий**.

Текстовий і двійковий режими представлення файлів. Розглянемо розбіжності, які існують між текстовим і двійковим вмістами файлу, текстовим і двійковим файловими форматами, а також текстовим і двійковим режимами для файлів.

Вміст усіх файлів зберігається в двійковій формі (нулі та одиниці). Але, якщо в файлі двійкові коди символів (наприклад, **ASCII**) використовуються головним чином для представлення тексту, майже як в

рядках **C**, то такий файл є текстовим, тобто має текстовий вміст. Якщо ж двійкові значення в файлі представляють код на машинній мові, числові дані (з застосуванням внутрішнього представлення значень, наприклад, **long** або **double**), кодування зображення або музичного твору, то вміст буде двійковим.

Щоб привнести деяку закономірність в обробку текстових файлів, мова **C** надає два способи доступу до файлів: двійковий режим і текстовий режим.

У двійковому режимі програмі доступний кожен байт файлу. Однак в текстовому режимі те, що бачить програма, може відрізнятися від того, що зберігається в файлі.

В текстовому режимі при читанні файлу представлення локального середовища для таких символів, як кінець рядка або кінець файлу, зіставляється з їх представленням в **C**.

Рівні вводу-виводу. Додатково до вибору представлення файлу в більшості випадків можна обрати один з двох рівнів вводу-виводу (тобто один з двох рівнів керування доступом до файлів). Низькорівневий ввід-вивід передбачає використання основних служб вводу-виводу, що надаються операційною системою. Стандартний високорівневий ввід-вивід передбачає застосування стандартного пакета бібліотечних функцій **C** та визначень з файлу заголовку **stdio.h**.

Стандарт **C** підтримує тільки стандартний пакет вводу-виводу, оскільки немає жодної можливості гарантувати, що усі операційні системи можуть бути представлені однаковою низькорівневою моделлю вводу-виводу.

Стандартні файли. Програми на **C** автоматично відкривають три файли, які називаються стандартним вводом (**stdin**), стандартним виводом (**stdout**) і стандартним виводом помилок (**stderr**). За замовчуванням стандартний ввід являє собою звичайний пристрій вводу у вашій системі, як правило, клавіатуру. Стандартний вивід і стандартний вивід помилок за замовчуванням є звичайним пристроєм виводу вашої системи, тобто екраном монітору.

Стандартний ввід забезпечує ввід даних до програми. Це файл, який читається за допомогою функцій **getchar()** і **scanf()**. Стандартний вивід місце, куди спрямовується звичайний вивід програми. Він забезпечується функціями **putchar()**, **puts()** і **printf()**. Призначення файлу стандартного виводу помилок полягає в тому, щоб надати логічно відокремлене місце для відправлення повідомлень про помилки.

8.1.2. Стандартний ввід-вивід

У порівнянні з низькорівневим вводом-виводом стандартний пакет вводу-виводу, разом з можливістю перенесення на інші реалізації, має ще дві переваги. По-перше, в ньому доступно багато спеціалізованих функцій, які спрощують рішення різних задач, що пов’язані з вводом-виводом. Наприклад, функція **printf()** перетворює різні форми даних у вивід рядків, що підходить для терміналів. По-друге, ввід і вивід є буферизованими. Це означає, що інформація передається великими порціями (зазвичай по **512** байтів і більше), а не по одному байту за один раз. Наприклад, коли програма читає файл, порція даних зчитується до буферу – проміжної області пам’яті. Така буферизація суттєво збільшує швидкість передачі даних. Потім програма може досліджувати окремі байти в цьому буфері. Буферизація відбувається приховано, тому створюється ілюзія посимвольного доступу.

В наступній програмі показано, як застосовувати стандартний ввід-вивід для запису та читання файлу, а також для підрахунку кількості символів, що знаходяться в ньому. Текст програми має такий вигляд:

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

int main(void)
{
    int ch;
    FILE *fp;                      // вказівник на файл
    unsigned long count = 0;
    char *name_file = "file1.txt";
    char words[] = "Програмування на С";
```

```
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251)

    if((fp = fopen(name_file, "w")) == NULL) {
      printf("Неможливо відкрити файл file1.txt\n");
      exit(EXIT_FAILURE);
    }
    else {
      fputs(words, fp);
    }
    fclose(fp);

    if((fp = fopen(name_file, "r")) == NULL)
    {
      printf("Неможливо відкрити файл file1.txt\n");
      exit(EXIT_FAILURE);
    }
    else {
      puts("Вміст файлу file1.txt\n");
      while((ch = getc(fp)) != EOF)
      {
          putc(ch, stdout); // те ж саме, що й putchar(ch);
          count++;
      }
      printf("\n\nКількість символів у файлі %s: %lu\n",
             name_file, count);
    }
    fclose(fp);

    return 0;
}
```

Результат виконання програми наведено на рис. 8.1.

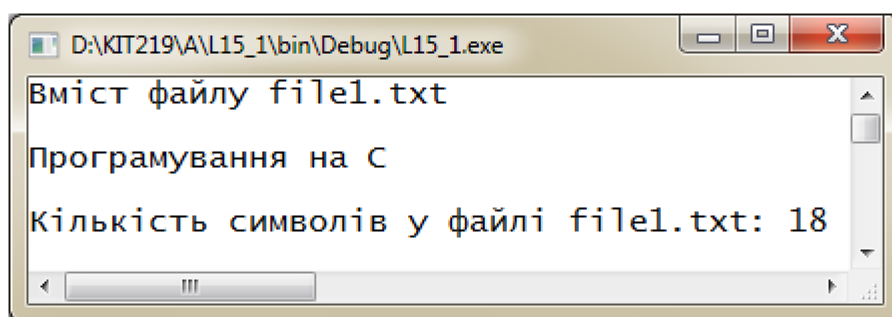


Рисунок 8.1 – Результат роботи програми, яка зчитує інформацію з файлу і підраховує кількість символів

Функція `fopen()`. Прототип функції `fopen()` має такий вигляд:

```
FILE *fopen(const char *fname, const char *mode);
```

Функція `fopen()` призначена для відкриття файлу. Вона оголошена в файлі заголовку `stdio.h`. Її першим аргументом є ім'я файлу, який необхідно відкрити (точніше, це адреса рядка, що містить ім'я файлу). Другий аргумент – рядок, що ідентифікує режим, в якому файл повинен бути відкритий. В бібліотеці `C` надається декілька можливостей застосування режимів, які зазначені в табл. 8.1.

Таблиця 8.1 – Рядки режиму відкриття файлу для функції `fopen()`

Рядок режиму	Опис
"r"	Відкрити текстовий файл для читання
"w"	Відкрити текстовий файл для запису з усіченням існуючого файлу до нульової довжини або створенням файлу, якщо він не існує
"a"	Відкрити текстовий файл для запису з додаванням даних у кінець існуючого файлу або створенням файлу, якщо він не існує
"r+"	Відкрити текстовий файл для оновлення (читання та запису)
"w+"	Відкрити текстовий файл для оновлення (читання та запису), попередньо зробивши усічення файлу до нульової довжини, якщо він існує, або, створивши файл, якщо його немає
"a+"	Відкрити текстовий файл для оновлення (читання та запису) з додаванням даних в кінець існуючого файлу або створенням файлу, якщо він не існує; читати можна весь файл, але записувати допускається тільки в кінець файлу
rb","wb", "ab","ab+", "a+b","wb+", "w+b","ab+", "a+b"	Подібні до попередніх режимів, за виключенням того, що замість текстового режиму вони використовують двійковий режим
"wx", "wbx", "w+x", "wb+x" або "w+bx"	(C11) Подібні до режимів без літери x, за виключенням того, що вони відмовляються працювати, якщо файл існує, та відкривають файл в монопольному режимі, якщо це можливо

Нові режими запису `C` з літерою `x` мають пару нових характеристик у порівнянні зі старими режимами запису. По-перше, якщо ви намагаєтесь відкрити в одному з традиційних режимів запису файл, що вже існує, то `fopen()` усікає файл до нульової довжини, в результаті чого попередній вміст файлу втрачається. Але режими з літерою `x` забезпечують в такому випадку

відмову функції **fopen()** від подальшої роботи, не причиняючи шкоди файлу. По-друге, за умови, що це дозволяє середовище, можливість монопольного доступу в режимах з **x** запобігає доступ до файлу з боку інших програм або потоків до тих пір, поки поточний процес не закриє цей файл.

Після успішного відкриття файлу функція **fopen()** повертає вказівник файлу, який потім інші функції вводу-виводу можуть використовувати для позначення цього файлу. Вказівник файлу (в програмі це **fp**) має тип вказівника на **FILE**, де **FILE** – похідний тип, що визначений у файлі **stdio.h**. Вказівник **fp** не посилається безпосередньо на файл. Замість цього він вказує на об'єкт даних, який містить інформацію про файл, включаючи відомості про буфер, що застосовується для файлового вводу-виводу. Оскільки функції вводу-виводу зі стандартної бібліотеки використовують буфер, їм необхідно знати, де цей буфер знаходиться. Їм також повинно бути відомо, наскільки заповнений буфер і з яким файлом здійснюється робота. Це дозволяє функціям за необхідності заповнювати або спустошувати буфер. Вся ця інформація міститься в об'єкті даних, на який вказує вказівник **fp**.

Функція **fopen()** повертає нульовий вказівник (також визначений в файлі **stdio.h**), якщо їй не вдається відкрити файл. Коли вказівник **fp** дорівнює значенню **NULL**, програма завершує своє виконання. Функція **fopen()** може відмовити у відкритті файлу через переповнення диску, відсутність файлу в заданому каталозі, неприпустиме ім'я, наявність україномовного або російськомовного шляху до файлу, обмеження доступу або апаратної проблеми. Це лише невеличка частина причин відмови.

Функції **getc()** і **putc()**. Прототипи функцій **getc()** і **putc()** мають такий вигляд:

```
int getc(FILE *stream);
int putc(int ch, FILE *stream);
```

Робота функцій **getc()** і **putc()** схожа з роботою функцій **getchar()** і **putchar()**. Відмінність полягає лише в тому, що цим новим

функціям треба вказати, з яким саме файлом їм необхідно працювати. Таким чином, наведений нижче оператор, який вже багато разів застосовувався раніше, означає «отримати символ зі стандартного вводу»:

```
ch = getchar();
```

Наступний оператор означає «отримати символ з файлу, на інформацію про який вказує **fp**»:

```
ch = getc(fp);
```

Аналогічним чином, наступний оператор означає «помістити символ **ch** у файл, на інформацію про який вказує **fpout**»:

```
putc(ch, fpout);
```

У списку аргументів **putc()** спочатку задається символ **ch**, а потім вказівник файлу **fpout**. В програмі, яка була розглянута раніше, у другому аргументі **putc()** застосовується **stdout**. Він визначений у файлі **stdio.h** як вказівник файлу, що асоціюється зі стандартним виводом, тому запис **putc(ch, stdout)** еквівалентний запису **putchar(ch)**. Насправді друга функція зазвичай визначена як перша. Аналогічно, **getchar()** визначена як функція **getc()**, яка використовує стандартний ввід.

Чому ж тоді в програмі використовується функція **putc()**, а не **putchar()**. Одна причина пов’язана з необхідністю ознайомлення з функцією **putc()**. Друга причина полягає в тому, що ви легко можете перетворити програму таким чином, щоб вона могла генерувати файловий вивід за рахунок використання аргументу, що відрізняється від **stdout**.

Кінець програми. Програма, яка читає дані з файлу, повинна зупинятися, коли вона досягає кінця файлу. Як можна повідомити програмі про те, що зустрівся кінець файлу?

Функція **getc()** повертає спеціальне значення **EOF**, якщо вона намагається прочитати символ і виявляє, що зустрівся кінець файлу. Таким чином, програма **C** дізнається про те, що вона досягла кінця файлу, тільки після

намагання прочитати за кінцем файлу.

Щоб уникнути проблем з намаганням читання порожнього файлу, під час файлового вводу повинен застосовуватися цикл з вхідною умовою. Через конструктивні особливості функції **getc()** (та інших функцій вводу **C**) програма повинна виконувати читання до входу в тіло циклу. Тоді певне підійде рішення, яке наведено нижче:

```
// рішення №1
int ch;          // змінна int для зберігання символів і EOF
FILE *fp;

fp = fopen("input.txt", "r");
ch = getc(fp);          // ввести перший символ з файлу
while(ch != EOF)
{
     putchar(ch);    // вивести поточний символ
     ch = getc(fp);  // отримати наступний символ
}
fclose(fp);
```

Це рішення можна вдосконалити наступним чином:

```
// рішення №2
int ch;
FILE *fp;
fp = fopen("input.txt", "r");
while((ch = getc(fp)) != EOF)
{
    putchar(ch);    // вивести поточний символ
}
fclose(fp);
```

Функція **fclose()** закриває файл, що ідентифікується вказівником **fp**, за необхідності звільняючи буфери. В більш відповідальній програмі треба переконатися, що файл закритий успішно. Функція **fclose()** повертає значення **0**, якщо файл був закритий успішно, і повертає **EOF**, якщо ні:

```
if(fclose(fp) != 0)
    printf("Помилка під час закриття файлу\n");
```

Функція **fclose()** може завершитися невдало, якщо, наприклад, жорсткий диск заповнений, зовнішній пристрій зберігання вилучено або

виникла помилка вводу-виводу.

Вказівники на стандартні файли. У **stdio.h** три вказівники файлів асоційовані з трьома стандартними файлами, які автоматично відкриваються програмами на **C** (табл. 8.2).

Таблиця 8.2 – Вказівники на стандартні файли

Стандартний файл	Вказівник файлу	Пристрій, що використовується
Стандартний ввід	**stdin**	клавіатура
Стандартний вивід	**stdout**	екран
Стандартний вивід помилок	**stderr**	екран

Усі вони мають тип вказівника на **FILE**, тому можуть використовуватися в якості аргументів для стандартних функцій вводу-виводу подібно до **fp** в програмі, що розглядалася раніше.

Наведемо приклад, в якому створюється новий файл та в який відбувається запис інформації. Програма копіює певні дані з одного файлу в інший. Вона відкриває два файли одночасно з використанням режиму **"r"** для одного та режиму **"w"** для другого. Текст програми, яка ущільнює вміст вхідного файлу, залишаючи тільки кожен третій символ, і поміщає ущільнений текст до другого файлу, має такий вигляд:

```
    #include <stdio.h>
    #include <windows.h>
    #include <stdlib.h>
#include <string.h>
    int main(void)
    {
        FILE *in, *out; // оголошення двох вказівників на FILE

        int ch;
        char *name_in  = "input.txt";
        char *name_out = "output.txt";

        int count = 0;

        SetConsoleCP(1251);
        SetConsoleOutputCP(1251);

        if((in = fopen(name_in, "r")) == NULL)
```

```
        {
          fprintf(stderr, "Неможливо відкрити файл input.txt\n");
          exit(EXIT_FAILURE);
        }
        else
        {
            if((out = fopen(name_out, "w")) == NULL)
            {
                printf("Неможливо відкрити файл output.txt\n");
                exit(EXIT_FAILURE);
            }
            else
            {
                while((ch = getc(in)) != EOF)
                {
                   if(count % 3 == 0)
                   putc(ch, out);
                    count++;
                }
            }
        }
        if(fclose(in) != 0 || fclose(out) != 0)
        fprintf(stderr, "Помилка під час закриття файлів\n");
        return 0;
    }
```

Для виконання цієї програми необхідно створити файл **input.txt** і занести до нього відповідну інформацію (рис. 8.2). Результат виконання програми наведено на рис. 8.3.

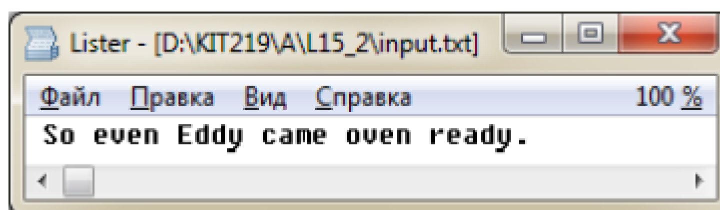


Рисунок 8.2 – Формування вмісту файлу для зчитування з нього інформації

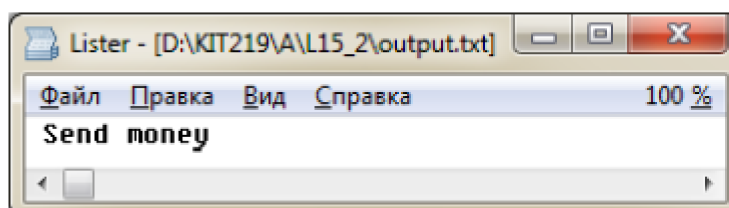


Рисунок 8.3 – Вміст вихідного файлу після вибіркового копіювання

Функція **fprintf()** подібна до функції **printf()** за виключенням того, що вона потребує передачі в якості першого аргумента вказівника файлу. В нашому випадку використовується вказівник **stderr** для відправки повідомлень про помилки до стандартного виводу помилок. Це стандартна практика в **C**.

В програмі є два одночасно відкритих файли, через що оголошені два вказівника на **FILE**. Зверніть увагу, що файли відкриваються та закриваються незалежно один від одного. Існує обмеження на кількість одночасно відкритих файлів, яке залежить від системи та реалізації. Часто це обмеження знаходиться в діапазоні від **10** до **20**. Один і той самий вказівник файлу можна використовувати для різних файлів за умови, що ці файли не відкриваються одночасно.

8.1.3. Файловий ввід-вивід: fprintf(), fscanf(), fgets()i fputs()

Для кожної функції вводу-виводу, які були розглянуті раніше, є схожа функція файлового вводу-виводу. Головна відмінність між ними полягає в тому, що функціям файлового вводу-виводу за допомогою вказівника на **FILE** необхідно повідомляти, з яким файлом їм працювати. Подібно до **getc()** і **putc()**, ці функції потребують ідентифікації файлу із застосуванням вказівника на **FILE**, такого як **stdout**, або значення, що повертається функцією **fopen()**.

Функції файлового вводу-виводу **fprintf()** і **fscanf()** працюють аналогічно до функцій **printf()** і **scanf()**. Різниця полягає тільки в наявності додаткового першого аргументу, в якому ідентифікується потрібний файл.

Ви вже застосовували раніше функцію **fprintf()**. В наступній програмі демонструється робота функцій файлового вводу-виводу поряд з функцією **rewind()**. Текст програми має такий вигляд:

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

```
#include <stdlib.h>
#include <string.h>
#define MAX 41

int main(void)
{
    FILE *fp;
    char words[MAX];

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251)
    ;

    if((fp = fopen("input.txt", "a+")) == NULL)
    {
        fprintf(stdout, "Не вдається відкрити файл" "
                          input.txt\n");
        exit(EXIT_FAILURE);
    }

    puts("Введіть слова для додавання до файлу.");
    puts("Для завершення введіть символ # на початку рядка.");
    while((fscanf(stdin,"%40s", words) == 1) &&
          (words[0] != '#'))
        fprintf(fp, "%s\n", words);
    puts("Вміст файлу:");

    rewind(fp);                  // повернення на початок файлу

    while(fscanf(fp,"%s",words) == 1)
        puts(words);
    puts("Готово!");
    if(fclose(fp) != 0)
        fprintf(stderr, "Помилка під час закриття файлу\n");
    return 0;
}
```

Ця програма дозволяє додавати слова до файлу. За рахунок використання режиму **"a+"** програма може здійснювати читання з файлу та запис інформації до файлу. При першому запуску вона створює файл **"input.txt"** і дозволяє поміщати до нього слова по одному в рядку. При подальшому запуску програма дозволяє додавати (дописувати) слова до існуючого вмісту. Режим додавання дозволяє тільки дописувати дані в кінець файлу, але режим **"a+"** дозволяє читати увесь файл.

Функція **rewind()** забезпечує переміщення на початок файлу, оскільки фінальний цикл **while** може вивести вміст файлу. Зверніть увагу,

що **rewind()** приймає вказівник файлу в якості аргументу.

Результат роботи програми наведено на рис. 8.4. Вміст файлу **input.txt** наведено на рис. 8.5.

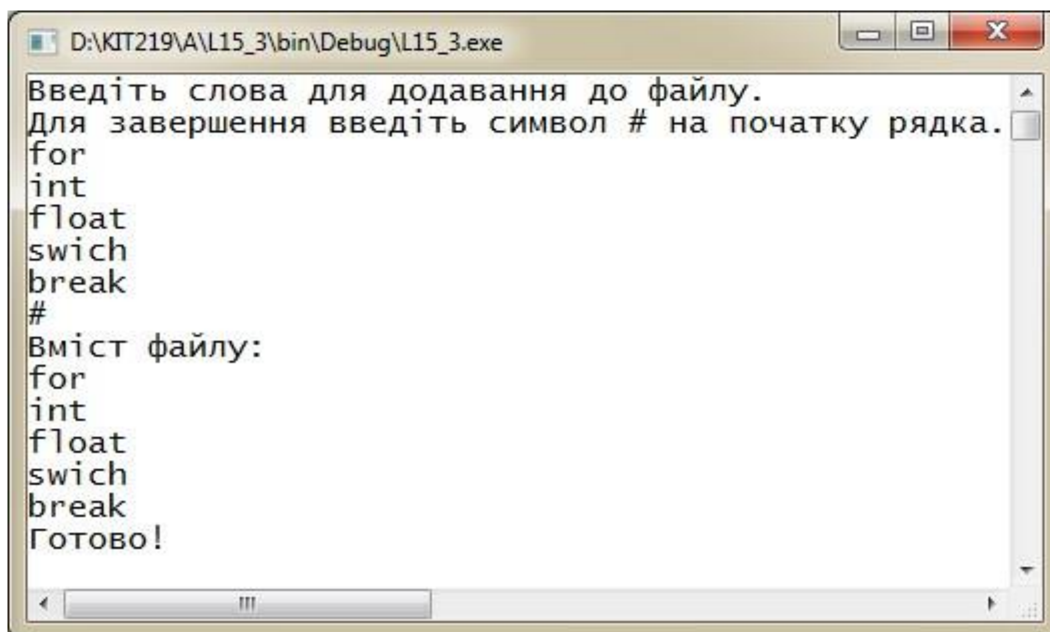


Рисунок 8.4 – Результат роботи програми, яка додає до файлу введені з клавіатури слова

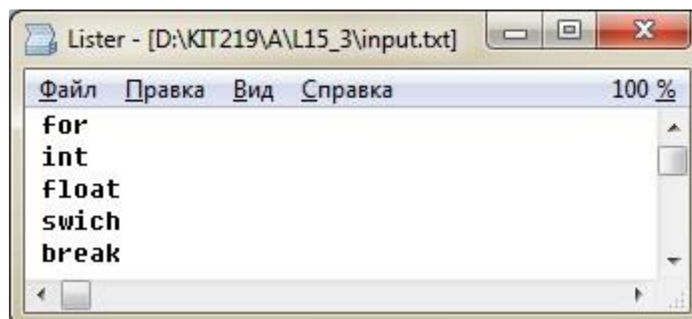


Рисунок 8.5 – Вміст файлу **input.txt**

Функції **fgets()** і **fputs()**. Першим аргументом функції **fgets()** є адреса (типу **char** *), де повинні зберігатися введені дані. Другий аргумент – ціле число, яке являє собою максимальний розмір вхідного рядка. Заключний аргумент – це вказівник файлу, який ідентифікує файл, що підлягає читанню. Виклик функції виглядає наступним чином:

```
fgets(buf, MAX, fp);
```

В даному операторі **buf** – це ім'я масиву типу **char**, **MAX** –

максимальний розмір рядка, а **fp** – вказівник на **FILE**.

Функція **fgets()** читає вхідні дані до появи першого символу нового рядка, тобто, до тих пір, поки не буде прочитана кількість символів, що на одиницю менше верхньої межі, або поки не буде виявлений кінець файлу. Потім **fgets()** додає завершальний нульовий символ, щоб сформувати рядок. Таким чином, верхня межа являє собою максимальну кількість символів плюс нульовий символ. Якщо **fgets()** вдасться прочитати цілий рядок до моменту досягнення граничного числа символів, вона розмістить символ нового рядка безпосередньо перед нульовим символом, позначивши кінець рядка. Функція **fgets()** повертає значення **NULL**, коли знаходить **EOF**. Цим можна скористатися для перевірки ознаки кінця файлу. В іншому випадку вона повертає адресу, яка була їй передана.

Функція **fputs()** приймає два аргументи: адресу рядка та вказівник файлу. Вона записує рядок, що знаходиться в зазначеній комірці, до визначеного файлу. На відміну від **puts()**, функція **fputs()** при виводі не додає символ нового рядка. Виклик **fputs()** виглядає наступним чином:

```
fputs(buf, fp);
```

В даному операторі **buf** є адресою ряжка, а **fp** ідентифікує цільовий файл. Оскільки **fgets()** зберігає символ нового рядка, а **fputs()** не додає цей символ, вони добре працюють в тандемі.

Функції довільного доступу до файлу fseek() і ftell(). Функція **fseek()** дозволяє трактувати файл подібно масиву і переходити безпосередньо до будь-якого байту в файлі, що відкритий за допомогою **fopen()**. Щоб ознайомитися з роботою **fseek()**, напишемо програму, яка відображає вміст вхідного файлу у зворотному порядку. Зверніть увагу на те, що функція **fseek()** приймає три аргументи і повертає значення **int**.

Функція **ftell()** повертає поточну позицію в файлі як значення **long**.

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

```
#define CNTL_Z '\032'    // маркер кінця файлу
#define SLEN 81
```

16

```
int main(void)
{
    char file[SLEN];
    char ch;
    FILE *fp;
    long count, last;

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    puts("Введіть ім'я файлу для обробки:");
    scanf("%80s", file);
    if((fp = fopen(file,"rb")) == NULL)
    {
        // режим тільки для читання
        printf("Файл неможливо відкрити %s\n", file);
        exit(EXIT_FAILURE);
    }
    fseek(fp, 0L, SEEK_END);  // перейти в кінець файлу
    last = ftell(fp);
    for(count = 1L; count <= last; count++)
    {
        // рухатися у зворотному напрямку
        fseek(fp, -count, SEEK_END);
        ch = getc(fp);
        if(ch != CNTL_Z && ch != '\r')  // файли MS DOS
            putchar(ch);
    }
    putchar('\n');
    fclose(fp);
    return 0;
}
```

Результат роботи програми наведено на рис. 8.6, а вміст файлу `input.txt` – на рис. 8.7.

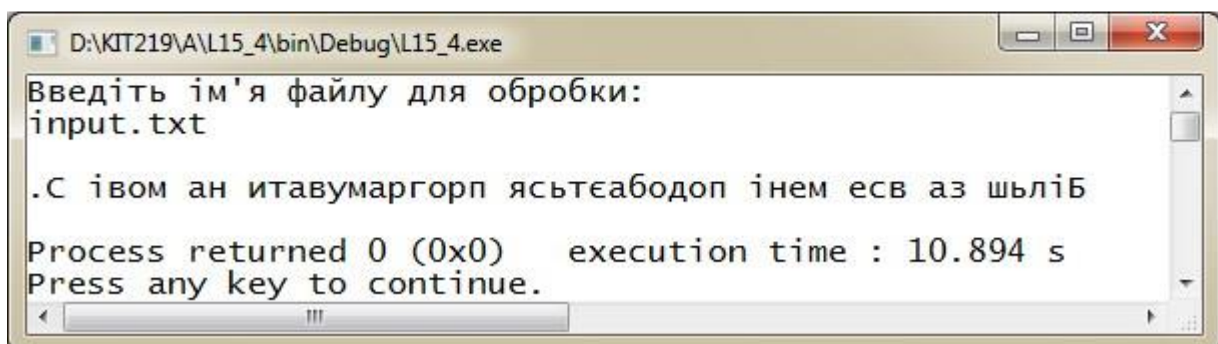


Рисунок 8.6 – Приклад застосування функцій `fseek()` і `ftell()`

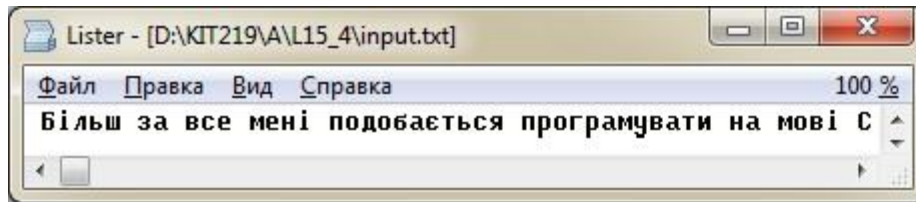


Рисунок 8.7 – Вміст файлу **input.txt**

Прототип функції **fseek()**:

```
int fseek(FILE *stream, long offset, int origin);
```

Першим з трьох аргументів функції **fseek()** є **stream** – вказівник типу **FILE** на файл, в якому буде відбуватися пошук. Файл повинен бути відкритий за допомогою функції **fopen()**.

Другий аргумент **offset** функції **fseek()** називається зміщенням. Він показує, наскільки далеко необхідно переміститися від стартової точки. В цьому аргументі необхідно передавати значення типу **long**, яке може бути додатним (переміститися вперед), від'ємним (переміститися назад) або нульовим (залишитися на місці).

Третій аргумент **origin** встановлює режим, що ідентифікує стартову точку. Починаючи зі стандарту **ANSI C**, у файлі заголовку **stdio.h** вказані іменовані константи для відповідних режимів (табл. 8.3).

Таблиця 8.3 – Режим, що ідентифікує стартову точку

Режим	Звідки вимірюється зміщення	Значення
SEEK_SET	Від початку файлу	0
SEEK_CUR	Від поточної позиції	1
SEEK_END	Від кінця файлу	2

Нижче наведені приклади виклику функції (**fp** – вказівник файлу):

```
fseek(fp, 0L, SEEK_SET); // перейти на початок файлу
fseek(fp, 10L, SEEK_SET); // вперед на 10 байтів від початку
fseek(fp, 2L, SEEK_CUR); // вперед на 2 байти від поточної позиції
```

```
fseek(fp, 0L, SEEK_END);   // перейти в кінець файлу
fseek(fp, -10L, SEEK_END); // назад на 10 байтів від кінця файлу
```

З такими викликами пов'язані можливі обмеження, про які мова буде йти далі. Значення, яке повертається функцією **fseek()**, дорівнює **0**, якщо все пройшло успішно, і **-1**, якщо виникла помилка, наприклад, намагання виходу за межі файлу.

Прототип функції **ftell()**:

```
long ftell(FILE *stream);
```

Функція **ftell()** має тип **long** і повертає поточну позицію у файлі. В стандарті **ANSI C** вона оголошена в файлі **stdio.h**. Функція **ftell()** вказує позицію в файлі, повертаючи кількість байтів від початку файлу, причому перший байт отримує номер **0**, другий – номер **1** і т. д. В **ANSI C** таке визначення застосовується до файлів, які відкриті у двійковому режимі, але не обов'язково до файлів, які відкриті в текстову режимі. Це одна з причин використання двійкового режиму у попередній програмі.

Розглянемо базові елементи програми. Перш за все, оператор

```
fseek(fp, 0L, SEEK_END);
```

встановлює позицію зі зміщенням **0** байтів від кінця файлу. Це означає встановлення позиції в кінець файлу. Потім оператор

```
last = ftell(fp);
```

присвоює **last** кількість байтів від початку до кінця файлу. Далі йде цикл:

```
for(count = 1L; count <= last; count++)
{
    // рухатися у зворотному напрямку
    fseek(fp, -count, SEEK_END);
    ch = getc(fp);
}
```

В першій ітерації відбувається позиціонування на перший символ перед кінцем файлу (тобто на фінальний символ у файлі). Після цього даний символ

виводиться. В наступній ітерації позиція встановлюється на передостанній символ у файлі, який потім виводиться. Процес продовжується до тих пір, поки не буде досягнутий і виведений перший символ у файлі.

Порівняння двійкового та текстового режимів. Багато редакторів **MS DOS** помічають кінець текстового файлу за допомогою символу **<Ctrl+Z>**. Коли такий файл відкривається в текстовому режимі в **C**, цей символ розглядається як ознака кінця файлу. Проте, коли той самий файл відкривається у двійковому режимі, **<Ctrl+Z>** є звичайним символом у файлі, а справжня ознака кінця файлу з'явиться пізніше. Він може знаходитися одразу після **<Ctrl+Z>** або ж файл може бути доповнений нульовими символами, щоб зробити розмір файлу кратним, скажімо, **256**. Нульові символи в середовищі **MS DOS** не відображаються, тому необхідно передбачити код, який запобіг би виводу символу **<Ctrl+Z>**.

Про іншу відмінність згадувалося раніше: символ нового рядка текстового файлу в **MS DOS** представлений за допомогою комбінації **\r\n**. Програма на **C**, яка відкриває той самий файл в текстовому режимі, «бачить» символи **\r\n** як просто **\n**, але у випадку застосування двійкового режиму програма бачить обидва символи, тобто **\r** і **\n**. У зв'язку з цим був застосований код для придушення виводу **\r**.

Функція **ftell()** може працювати по-різному в текстовому та двійковому режимах. В стандарті **ANSI C** стверджується, що для текстового режиму **ftell()** повертає значення, яке може бути використане в якості другого аргументу **fseek()**. Наприклад, в **MS DOS** функція **ftell()** може повертати кількість, при обчисленні якої, комбінація **\r\n** розглядається як один байт.

Перенесення коду з **fseek()** *і* **ftell()** *на інші реалізації.* Стандарт **ANSI C** передбачає для функцій **fseek()** і **ftell()** зменшені очікування. Далі описані деякі обмеження.

1) У двійковому режимі реалізації не обов'язково будуть підтримувати

режим **SEEK_END**. Тому перенесення коду попередньої програми не гарантоване. Більш зручний підхід передбачає читання всього файлу байт за байтом, поки не зустрінеться кінець. Але послідовне читання для знаходження кінця файлу є більш повільним, ніж просто перехід в його кінець. Директиви умовної компіляції препроцесора **C**, які будуть обговорюватися в одній з подальших лекцій, пропонують більш систематизований спосіб для підтримки вибору альтернативного коду.

2) В текстовому режимі будуть гарантовано працювати тільки виклики функції **fseek()**, які наведені в табл. 8.4.

Таблиця 8.4 – Виклики функції **fseek()**, які точно будуть працювати

Виклик функції	Результат
`fseek(file, 0L, SEEK_SET);`	Перейти в початок файлу
`fseek(file, 0L, SEEK_CUR);`	Залишатися у поточній позиції
`fseek(file, 0L, SEEK_END);`	Перейти в кінець файлу
`fseek(file, ftell_pos, SEEK_SET);`	Перейти в позицію ftell pos від початку файлу; ftell_pos - це значення, що повертається функцією ftell()

Одна потенційна проблема з функціями **fseek()** і **ftell()** полягає в тому, що вони обмежують розміри файлів значеннями, які можуть бути представлені типом **long**. Можливо, **2** мільярди байтів можуть здатися більш ніж достатнім розміром, але об’єми пристроїв зберігання, що постійно зростають, дозволяють працювати з файлами більших розмірів. В **ANSI C** з’явилися дві нові функції позиціонування, які спроектовані для роботи з файлами великих розмірів. Замість застосування для представлення позиції значення **long**, вони використовують новий тип **fpos_t** (від **file position type** – тип для позначення позиції в файлі). Тип **fpos_t** не є фундаментальним, а визначається в термінах інших типів. Змінна або об’єкт даних типу **fpos_t** може вказувати позицію всередині файлу і не може бути масивом, але

його природа подібного і не вимагає. Реалізація може надати будь-який тип, що задовольняє потребам конкретної платформи. Такий тип може бути реалізований, наприклад, у вигляді структури.

В **ANSI C** визначено, як застосовувати тип **fpos_t**. Функція **fgetpos()** має такий прототип:

```
int fgetpos(FILE *restrict stream, fpos_t *restrict pos);
```

Виклик **fgetpos()** поміщає поточне значення типу **fpos_t** в комірку, яка вказується вказівником **pos**. Це значення описує позицію в файлі. Функція повертає нуль у випадку успіху та ненульове значення у випадку відмови.

Прототип функції **fsetpos()** виглядає наступним чином:

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

Виклик **fsetpos()** призводить до використання значення типу **fpos_t** з комірки, яка задається за допомогою **pos**, для встановлення вказівника файлу в позицію, що містить це значення. Функція повертає нуль у випадку успіху та ненульове значення у випадку відмови. Значення **fpos_t** повинне бути отримане попереднім викликом функції **fgetpos()**.

Принцип роботи стандартного вводу-виводу. Зазвичай першим кроком у застосуванні стандартного вводу-виводу є виклик функції **fopen()** для відкриття файлу.

Функція **fopen()** не тільки відкриває файл, але й налаштовує буфер (або два буфери для режимів читання-запису) і встановлює структуру даних, яка містить відомості про файл і про буфер. Крім того, **fopen()** повертає вказівник на цю структуру, так що інші функції знають, де її шукати. Припустимо, що це значення присвоєно змінній **fp** типу вказівника. Говорять, що функція **fopen()** «відкриває потік даних». Якщо файл відкривається в текстовому режимі, ви отримуєте текстовий потік, а якщо у двійковому – двійковий потік.

Ця структура даних зазвичай містить індикатор, що призначений

для визначення поточної позиції у файлі. Вона також містить індикатори для помилок і кінця файлу, вказівник на початок буфера, ідентифікатор файлу та лічильник кількості байтів, які були скопійовані до буфера.

Зосередимо увагу на файловому вводі. Зазвичай наступним кроком є виклик однієї з функцій вводу, що оголошені в файлі **stdio.h**, таких як **fscanf()**, **getc()** або **fgets()**. Виклик будь-якої такої функції призводить до того, що порція даних копіюється з файлу до буфера. Розмір буфера залежить від реалізації, але зазвичай він має **512** або кратну до цього кількість байтів, таку як **4096** або **16384**. У зв'язку зі збільшенням об'ємів жорстких дисків і пам'яті комп'ютера, розміри буферів, які обираються, також мають тенденцію до зростання. Додатково до заповнення буфера початковий виклик функції встановлює значення в структурі, яка вказується за допомогою **fp**. Зокрема, встановлюється поточна позиція в потоці даних і кількість байтів, які скопійовані до буферу. Зазвичай поточна позиція починається з байту **0**.

Після ініціалізації структури даних і буфера, функція вводу читає за вимогою дані з буфера. В результаті індикатор позиції встановлюється таким чином, щоб вказати на символ, який йде наступним за останнім прочитаним символом. Оскільки усі функції вводу з сімейства **stdio.h** використовують такий самий буфер, виклик будь-якої такої функції поновлює читання там, де воно було припинено попереднім викликом будь-якої з функцій.

Коли функція вводу з'ясовує, що усі символи з буфера прочитані, вона робить запит на копіювання з файлу до буфера наступної порції даних з об'ємом, який дорівнює розміру буфера. Таким чином функції вводу можуть читати увесь вміст файлу. Після того, як функція прочитає останній символ фінальної порції даних, вона встановлює індикатор кінця файлу в істинне значення. Наступний виклик будь-якої функції вводу поверне **EOF**.

Функції виводу в схожому стилі здійснюють запис до буферу. Коли буфер заповнюється, дані копіюються до файлу.

8.1.4. Інші стандартні функції вводу-виводу

Стандартна бібліотека **ANSI C** містить більше трьох десятків функцій, що утворюють сімейство для стандартного вводу-виводу. Коротко опишемо ще декілька функцій, для кожної з яких буде наданий прототип **C**, який зазначає її аргументи та значення, що повертається. Усі функції, про які йде мова (крім **setvbuf()**), доступні в реалізаціях, які передують **ANSI C**.

Функція **ungetc()** Прототип функції:

```
int ungetc(int c, FILE *fp);
```

Функція **ungetc()** повертає символ, що вказаний в **c**, назад до вхідного потоку. У випадку повернення символу до вхідного потоку він буде прочитаний наступним викликом стандартної функції вводу (рис. 8.8).

Рисунок 8.8 – Принцип роботи функції **ungetc()**

Припустимо, наприклад, що потрібна функція, яка читає усі символи до наступного символу **":"**, не враховуючи його. Можна застосувати **getchar()** або **getc()** для читання символів до двокрапки і потім викликати **ungetc()**, щоб повернути двокрапку назад до вхідного потоку. Стандарт **ANSI C** гарантує тільки одне повернення символу за один раз. Якщо реалізація дозволяє повертати одразу декілька символів у рядку, функції вводу прочитають їх у зворотному порядку.

Прототип функції **fflush()** виглядає наступним чином:

```
int fflush(FILE *fp);
```

Виклик функції **fflush()** призводить до того, що будь-які дані в буфері виводу, які ще не були записані, відправляються у вихідний файл, що ідентифікується за допомогою **fp**. Цей процес називається скиданням буфера. Якщо **fp** – нульовий вказівник, то скидаються усі буфери виводу. Результат використання функції **fflush()** на вхідному потоці є невизначеним. Функцію **fflush()** можна застосовувати з потоком оновлення (для будь-якого режиму читання-запису), за умови, що найостанніша операція, яка використовувала потік, не була операцією вводу.

Прототип функції **setvbuf()** має такий вигляд:

```
int setvbuf(FILE *restrict fp, char *restrict
            buf, int mode, size_t size);
```

Функція **setvbuf()** встановлює альтернативний буфер, що призначений для застосування стандартними функціями вводу-виводу. Вона викликається після того, як файл був відкритий, і перед виконанням будь-якої іншої операції з потоком даних. Вказівник **fp** ідентифікує потік, а **buf** вказує на місце зберігання. Значення **buf**, яке не дорівнює **NULL**, говорить про те, що буфер створюється самостійно. Наприклад, можна оголосити масив з **1024** елементів типу **char** і передати адресу цього масиву. Однак якщо в якості значення **buf** вказується **NULL**, то функція сама виділяє пам'ять під буфер. Аргумент **size** повідомляє **setvbuf()** розмір цього масиву.

Для **mode** доступні наступні варіанти:

_IOFBF означає повну буферизацію (буфер скидається, коли він повний);

_IOLBF означає порядкову буферизацію (буфер скидається, коли він повний або коли в нього записаний символ нового рядка);

_IONBF означає відсутність буферизації.

Функція повертає нуль у разі успіху і ненульове значення в іншому випадку. Припустимо, що існує програма, яка працює зі збереженими

об’єктами даних, що мають розмір, скажімо, по 3000 байтів кожний. Тоді можна за допомогою функції `setvbuf()` створити буфер, розмір якого кратний розміру об’єкта даних.

8.2. Двійковий файловий ввід-вивід

8.2.1. Двійковий ввід-вивід: функції fread()і fwrite()

Стандартні функції вводу-виводу, які ви застосовували до сих пір, працюючи з символами та рядками, були орієнтовані на текст. Якщо в файлі треба зберегти числові дані, можна скористатися функцією `fprintf()` і форматом %f, щоб зберегти значення з рухомою комою, але тоді воно зберігається як послідовність символів.

Найбільш точний спосіб зберігання числа передбачає використання того ж самого набору бітів, що й використовує комп’ютер. Таким чином, значення типу `double` повинне бути збережено в області пам’яті, яка має розмір достатній для зберігання типу `double`. Коли дані зберігаються в файлі у вигляді, який застосовується у програмі, ми говоримо, що дані зберігаються у двійковій формі. Жодних перетворень з числових форм у послідовності символів не відбувається. Для стандартного вводу-виводу таку послугу пропонують функції `fread()` і `fwrite()`, робота яких ілюструється на рис. 8.9.

В дійсності усі дані зберігаються в двійковій формі. Навіть символи зберігаються з використанням двійкового представлення їх кодів. Однак якщо усі дані в файлі інтерпретуються як коди символів, говорять, що файл містить текстові дані. Якщо деякі або усі дані інтерпретуються як числові дані в двійковій формі, говорять, що файл містить двійкові дані. Крім того, двійковими також є файли, в яких дані являють собою команди на машинній мові.

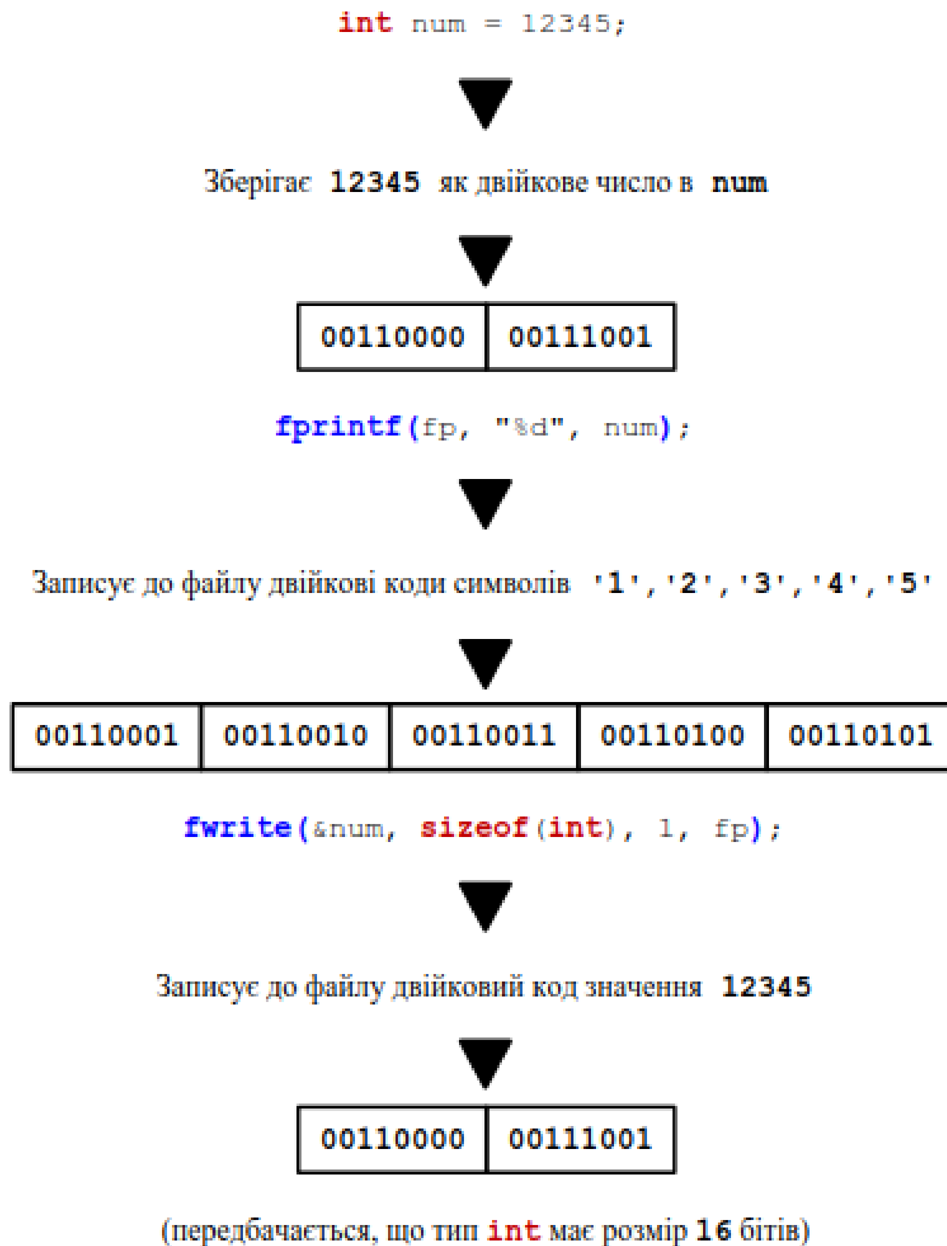


Рисунок 8.9 – Принцип роботи функцій **fprintf()** і **fwrite()**

Застосування термінів «двійковий» і «текстовий» може призвести до плутанини. Стандарт **ANSI C** розрізнює два режими відкриття файлів: двійковий і текстовий. Більшість з операційних систем розрізняють два файлові формати: двійковий і текстовий. Усі ці характеристики пов’язані, але не є ідентичними. Ви можете відкривати файл текстового формату у двійковому режимі. Ви можете зберігати текст у файлі двійкового формату. Ви

можете використовувати функцію getc() для копіювання файлів, що містять двійкові дані. Однак для зберігання двійкових даних у файлі двійкового формату ви зазвичай будете застосовувати двійковий режим. Аналогічно, частіше за все робота з текстовими даними в текстових файлах відбувається при їх відкритті в текстовому режимі. Файли, що генеруються текстовими процесорами, як правило, є двійковими, оскільки вони містять багато нетекстової інформації, яка описує шрифти та форматування.

8.2.2. Функція fwrite().

Нижче показано прототип функції fwrite():

```
size_t fwrite(const void * restrict ptr, size_t
              size, size_t nmemb, FILE * restrict
              fp);
```

Функція fwrite() записує двійкові дані до файлу. Тип size_t визначений в термінах стандартних типів C. Це тип, що повертається операцією sizeof. Зазвичай їм є тип unsigned int, але реалізації можуть обирати інший тип. Вказівник ptr – це адреса порції даних, що призначена для запису. Аргумент size являє собою розмір в байтах порції даних, що підлягають запису, а nmemb – кількість таких порцій.

Як завжди, fp ідентифікує файл, в який повинен проводитись запис. Наприклад, щоб зберегти об'єкт даних (такий як масив) розміром 256 байтів, можна вдіяти таким чином:

```
char buffer[256];
fwrite(buffer, 256, 1, fp);
```

Цей виклик fwrite() записує одну порцію даних розміром 256 байтів з буфера до файлу. Щоб зберегти, скажімо, масив з 10 елементів типу double, знадобляться наступні оператори:

```
double mas[10];
fwrite(mas, sizeof(double), 10, fp);
```

Цей виклик `fwrite()` записує дані з масиву `mas` до файлу `10` порціями даних, кожна з яких має розмір `double`.

Можливо, ви звернули увагу на дивне оголошення

```
const void * restrict ptr
```

в прототипі `fwrite()`. Проблема, що пов'язана з функцією `fwrite()`, полягає в тому, що її перший аргумент не має фіксованого типу. Скажімо, в першому прикладі використовувався аргумент `buffer`, що має тип вказівника на `char`, а в другому прикладі – аргумент `mas` з типом вказівника на `double`. В контексті прототипів **ANSI C** ці фактичні аргументи перетворюються на тип вказівника на тип `void`, який діє як свого роду універсальний тип для вказівників.

Функція `fwrite()` повертає кількість успішно записаних елементів. Зазвичай вона дорівнює `nmemb`, однак може бути меншою, якщо виникла помилка запису.

8.2.3. Функція fread()

Прототип функції `fread()` має такий вигляд:

```
size_t fread(void * restrict ptr, size_t size,
             size_t nmemb, FILE * restrict fp);
```

Функція `fread()` приймає такий самий набір аргументів, як і `fwrite()`. Цього разу `ptr` являє собою адресу області пам'яті, куди поміщаються дані, які прочитані з файлу, а `fp` ідентифікує файл, з якого відбувається читання.

Цю функцію слід використовувати для читання даних, які були записані до файлу за допомогою `fwrite()`. Наприклад, для того щоб відновити масив з `10` елементів типу `double`, який був збережений у попередньому прикладі, слід застосувати такий код:

```
double mas[10];
fread(mas, sizeof(double), 10, fp);
```

Цей виклик копіює `10` значень розміру **`double`** до масиву **`mas`**.

Функція **`fread()`** повертає кількість успішно прочитаних елементів. Зазвичай вона дорівнює **`nmemb`**, однак може бути й меншою, якщо виникла помилка запису або був досягнутий кінець файлу.

8.2.4. Функції feof() і ferror()

Коли стандартні функції вводу повертають **`EOF`**, це зазвичай означає, що досягнутий кінець файлу. Тим не менш, повернення **`EOF`** може також вказувати на виникнення помилки читання. Функції **`feof()`** і **`ferror()`** дозволяють проводити різницю між цими двома можливостями.

Функція **`feof()`** повертає ненульове значення, якщо при останньому виклику функції вводу був виявлений кінець файлу, і нуль – в іншому випадку.

Функція **`ferror()`** повертає ненульове значення, якщо виникла помилка читання або запису, і нуль – в іншому випадку.

Розглянемо приклад використання функцій **`fread()`** і **`fwrite()`**.

Давайте скористаємося деякими з цих функцій в програмі, яка додає вміст зі списку файлів у кінець вказаного файлу. Одна з задач полягає в передачі всередині програми інформації про файли. Це можна робити інтерактивно або за допомогою аргументів командного рядка. Застосуємо перший підхід, який передбачає виконання перерахованих нижче дій:

1) Запит імені файлу призначення та його відкриття; 2) Застосування циклу для запиту вхідних файлів;

3) Почергове відкриття кожного вхідного файлу в режимі читання та додавання його вмісту в кінець файлу призначення.

Щоб проілюструвати роботу функції **`setvbuf()`**, ми застосуємо її для встановлення іншого розміру буфера. Наступний етап деталізації пов’язаний з відкриттям файлу призначення. Ми будемо використовувати наступні кроки:

1) Відкрити файл призначення в режимі додавання;

2) Якщо цього зробити неможна, то завершити роботу;

3) Встановлення буферу розміром **4096** байтів для цього файлу; 4) Якщо цього зробити неможна, то завершити роботу.

Аналогічно, ми можемо уточнити частину програми, що відповідає за копіювання, для чого виконати з кожним файлом такі дії:

1) Якщо це файл призначення, то пропустити його та перейти до наступного файлу;

2) Якщо файл не може бути відкритий в режимі читання, то пропустити його та перейти до наступного файлу;

3) Додати вміст файлу до файлу призначення.

На завершення програма перейде на початок файлу призначення та відобразить його вміст. З практичних міркувань для копіювання будуть застосовувалися функції **fread()** і **fwrite()**. Текст програми має такий вигляд:

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

#define BUFSIZE 4096
#define SLEN   81
void append(FILE *source, FILE *dest);
char *s_gets(char *st, int n);
int main(void)
{
  FILE *fs; // fs для вхідного файлу
  FILE *fa; // fa для файлу призначення
  int files = 0; // кількість файлів, що додаються
  char file_app[SLEN]; // ім'я файлу призначення
  char file_src[SLEN]; // ім'я вхідного файлу
  int ch;

  SetConsoleOutputCP(1251);

  puts("Введіть і'мя файлу призначення:");
  s_gets(file_app, SLEN);
  if((fa = fopen(file_app, "a+")) == NULL)
  {
    fprintf(stderr, "Не вдається відкрити %s\n", file_app);
    exit(EXIT_FAILURE);
```

```
    }
    if(setvbuf(fa, NULL, _IOFBF, BUFSIZE) != 0)
    {
      fputs("Не вдається створити вихідний буфер\n", stderr);
      exit(EXIT_FAILURE);
    }
    puts("\nВведіть ім'я першого файлу "
         "(або пустий рядок для завершення):");
    while(s_gets(file_src, SLEN) && file_src[0] != '\0')
    {
      if(strcmp(file_src, file_app) == 0)
        fputs("Додати файл у кінець самого себе неможливо",
        stderr);
    else
    {
    if((fs = fopen(file_src, "r")) == NULL)
       fprintf(stderr, "Не вдається відкрити %s\n",
               file_src);
    else
    {
       if(setvbuf(fs, NULL, _IOFBF, BUFSIZE) != 0)
       {
          fputs("Не вдається створити вхідний буфер\n",stderr);
          continue;
       }
        append(fs, fa);
        if(ferror(fs) != 0)
          fprintf(stderr,"Помилка при читанні файлу %s.\n",
                  file_src);
        if(ferror(fa) != 0)
          fprintf(stderr, "Помилка при запису файлу %s.\n",
                  file_app);
        fclose(fs);
        files++;
       printf("Вміст файлу %s додано.\n", file_src);
       puts("Введіть ім'я наступного файлу "
              "(або порожній рядок для завершення):");
    }
    }
  }
  printf("Додавання завершено.\n");
  printf("Кількість доданих файлів: %d.\n\n", files);
  rewind(fa);
  printf("Вміст %s:\n", file_app);
  puts("====================================");
  while((ch = getc(fa)) != EOF)
  putchar(ch); puts("\n====================================");
  puts("Задачу завершено.");
  puts("====================================\n"); fclose(fa);
  return 0;
}

void append(FILE *source, FILE *dest)
```

```
{
    size_t bytes;
    static char temp[BUFSIZE]; // виділити пам'ять один раз
    fputs("\n", dest);
    while((bytes = fread(temp,sizeof(char),BUFSIZE,source)) > 0)
      fwrite(temp, sizeof(char), bytes, dest);
}

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 є NULL,
       *find = '\0';          // помістити туди нульовий символ
      else
        while(getchar()
          continue;
    }
    return ret_val;
}
```

Для роботи даної програми необхідно створити декілька файлів і наповнитиїх вмістом. В нашому випадку були створені 2 файли з іменами `input1.txt` (рис. 8.10) и `input2.txt` (рис. 8.11).

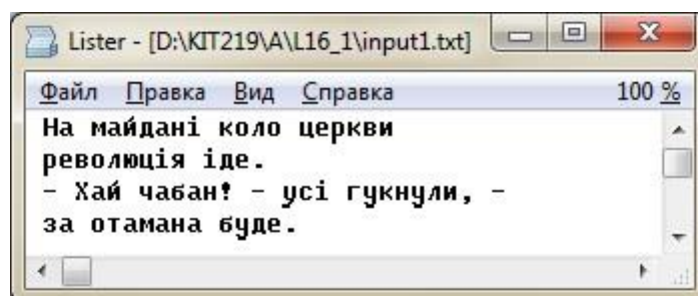


Рисунок 8.10 – Вміст файлу `input1.txt`

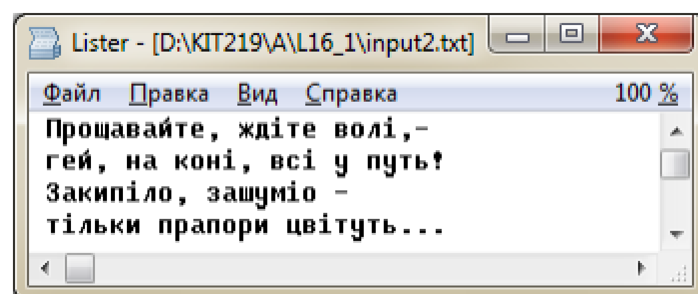


Рисунок 8.11 – Вміст файлу `input2.txt`

Результат виконання програми наведено на рис. 8.12.

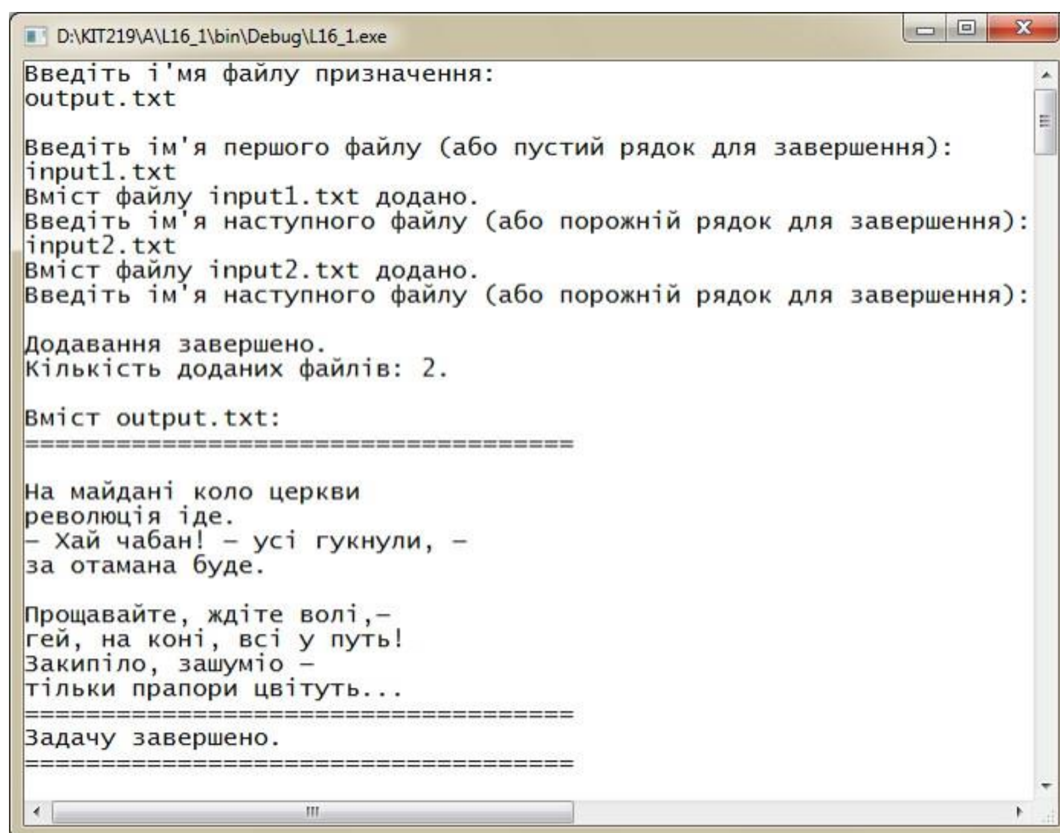


Рисунок 8.12 – Результат виконання програми

Вміст вихідного файлу **output.txt** має такий вигляд (рис. 8.13):

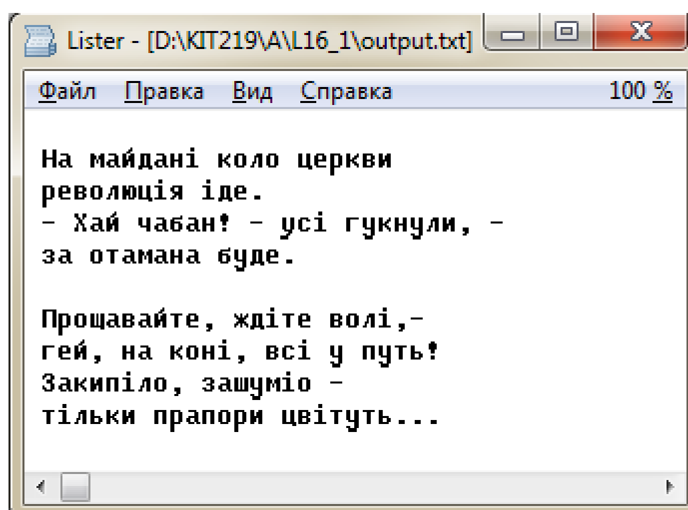


Рисунок 8.13 – Вміст файлу **output.txt**

Якщо функція **setvbuf()** буде не в змозі створити буфер, вона поверне ненульове значення, після чого програма припинить роботу. Схожий код встановлює буфер розміром **4096** байтів для файлу, що

копіюється в поточний момент. За рахунок використання **NULL** в другому аргументі **setvbuf()** ми дозволяємо цій функції самостійно виділити пам'ять під буфер.

Для отримання імені файлу в програмі застосовується функція **s_gets()** замість **scanf()**, оскільки **scanf()** пропускає пробільні символи і, відповідно, не зможе знайти порожній рядок. Крім того, в програмі використовується **s_gets()** замість простої функції **fgets()**,тому що **fgets()** залишає в рядку символ нового рядка.

Код, що показаний нижче, запобігає додаванню вмісту файлу в кінець самого себе:

```
if(strcmp(file_src, file_app) == 0)
  fputs("Додати файл у кінець самого себе неможливо", stderr);
```

Аргумент **file_app** являє собою ім'я файлу призначення, а **file_src** – ім'я файлу, що обробляється в поточний момент.

Функція **append()** виконує копіювання. Замість копіювання по одному байту за один раз вона застосовує **fread()** і **fwrite()** для копіювання по **4096** байтів за один раз:

```
void append(FILE *source, FILE *dest)
{
  size_t bytes;
  static char temp[BUFSIZE]; // виділити пам'ять один раз

  fputs("\n", dest);
  while((bytes = fread(temp,sizeof(char),BUFSIZE,source)) > 0)
    fwrite(temp, sizeof(char), bytes, dest);
}
```

Оскільки файл, що вказаний як **dest**, відкритий в режимі додавання, вміст вхідних файлів по черзі додається в кінець файлу **dest**. Зверніть увагу, що масив **temp** має статичну тривалість зберігання (це означає, що пам'ять під нього виділяється на етапі компіляції, а не кожного разу, коли викликається **append()**) і область видимості в межах блока (тобто він є закритим для даної функції).

В прикладі використовуються файли в текстовому режимі. Шляхом застосування режимів **"ab+"** і **"rb"** можна було б обробляти двійкові файли.

8.2.5. Довільний доступ з двійковим вводом-виводом

Довільний доступ частіше за все застосовується до двійкових файлів, що записані з використанням двійкового вводу-виводу. Розглянемо програму, яка створить файл з числами типу **double** і потім надасть доступ до його вмісту.

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#define ARSIZE 1000

int main(void)
{
  double numbers[ARSIZE];
  double value;
  const char *file = "numbers.dat";
  int i;
  long pos;
  FILE *iofile;
  SetConsoleOutputCP(1251);

  // створення набору значень типу double
  for(i = 0; i < ARSIZE; i++)
    numbers[i] = 100.0 * i + 1.0 / (i + 1);
  // спроба відкрити файл
  if((iofile = fopen(file, "wb")) == NULL)
  {
  fprintf(stderr,"Не вдається відкрити файл %s для виводу\n",
          file);
  exit(EXIT_FAILURE);
}

// запис у файл масиву в двійковому форматі
fwrite(numbers, sizeof(double), ARSIZE, iofile);
fclose(iofile);
if((iofile = fopen(file, "rb")) == NULL)
{
    fprintf(stderr, "Не вдається відкрити файл %s для" "
                    довільного доступу.\n", file);
    exit(EXIT_FAILURE);
}
// читання обраних елементів з файлу
printf("Введіть індекс в діапазоні 0-%d.\n", ARSIZE - 1);
while(scanf("%d", &i) == 1 && i >= 0 && i < ARSIZE)
{
   pos = (long)i * sizeof(double); // обчислення зміщення
   fseek(iofile, pos, SEEK_SET);   // перехід в необхідну
```

```
                                             // позицію
    fread(&value, sizeof(double), 1, iofile);
    printf("За цим індексом знаходиться значення %f\n", value);
    printf("\nВведіть наступний індекс\n(або значення"
            " за межами діапазону для завершення):\n");
  }
  // завершення
  fclose(iofile);
  puts("Програма завершена.");
  return 0;
}
```

Результат виконання програми наведено на рис. 8.14.

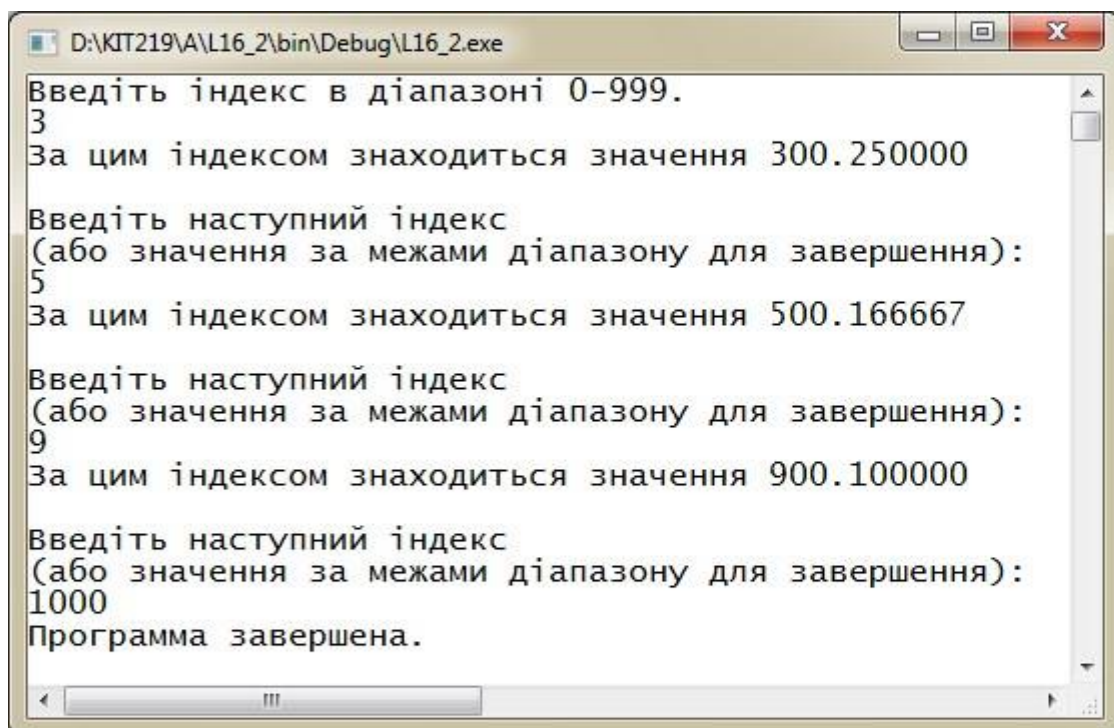


Рисунок 8.14 – Результат виконання програми для довільного доступу до елементів масиву, який заповнений випадковими дійсними числами

Спершу програма створює масив і поміщає в нього ряд значень типу **double**. Потім вона створює файл по імені **numbers.dat** в двійковому режимі та застосовує функцію **fwrite()** для копіювання вмісту масиву до цього файлу. **64**-бітова послідовність для кожного значення **double** копіюється з пам'яті у файл. Ви не зможете прочитати результуючий двійковий файл в текстовому редакторі, оскільки ці значення не транслюються в рядки. Однак кожне значення зберігається в файлі так само, як воно зберігалося в пам'яті, тому точність не втрачається. Більш того, кожне

значення займає 64 біта простору в файлі, завдяки чому легко обчислювати місцезнаходження кожного значення.

В другій частині програми файл відкривається для читання і користувачу пропонується ввести індекс значення. Множення значення індексу на кількість байтів, яку займає тип `double`, дає позицію в файлі. Далі в програмі викликається функція `fseek()` для переходу в цю позицію і `fread()` для читання значення з цього місця. Зверніть увагу на відсутність специфікаторів формату. Замість цього `fread()` копіює 8 байтів, починаючи з заданої позиції, до комірки пам’яті, що вказана в `&value`. Після цього програма використовує функцію `printf()` для відображення `value`.

8.2.6. Ключові поняття

Програма `C` розглядає ввід як потік байтів. Джерелом цього потоку може бути файл, пристрій вводу (наприклад, клавіатура) або навіть вивід з іншої програми. Подібним чином програма `C` трактує вивід як потік байтів. Місцем призначення може бути файл, екран монітора та інше.

Те, як в `C` інтерпретується вхідний або вихідний потік байтів, залежить від функцій вводу-виводу, що застосовуються. Програма може читати та зберігати байти без їх змін або інтерпретувати байти як символи, які, в свою чергу, можуть бути інтерпретовані як звичайний текст або текстове представлення чисел.

Аналогічно, при виводі функції, що використовуються, визначають, чи передаються двійкові значення без змін або перетворюються на текст або текстове представлення чисел. Якщо є числові дані, які ви бажаєте зберегти і потім відновити без втрати точності, застосовуйте двійковий режим і функції `fread()` і `fwrite()`. Якщо ви зберігаєте текстову інформацію та бажаєте створити файл, який може бути переглянутий за допомогою звичайних текстових редакторів, використовуйте текстовий режим і такі функції, як `getc()` і `fprintf()`.

Для доступу до файлу вам необхідно створити вказівник файлу (типу

FILE *) і зв'язати його з конкретним іменем файлу. Для роботи з файлом в подальшому коді буде застосовувати цей вказівник, а не ім'я файлу.

Важливо розуміти яким чином в **C** підтримується концепція кінця файлу. Зазвичай у програмі використовується цикл для читання вхідних даних до тих пір, поки не буде досягнуто кінця файлу. Функції вводу **C** не виявляють кінець файлу до тих пір, поки вони не будуть намагатися читати за кінцем файлу. Це означає, що перевірка щодо кінця файлу повинна відбуватися безпосередньо після намагання прочитати дані.

Контрольні запитання та завдання

1. Що таке файл?
2. У чому полягають особливості текстових файлів?
3. В чому перевага використання файлів у порівнянні з масивами?
4. Які операції можна проводити з файлами?
5. Яким чином відбувається запис до файлу?
6. Як відбувається читання з файлу?
7. Як отримати доступ до елементу файлу з заданим номером?
8. Як закрити файл та чому це потрібно робити?
9. Чи може файл складатися тільки з одного елемента?
10. Який файл називається текстовим?
11. Який файл називається бінарним?
12. Чи можуть рядки в текстових файлах мати різну довжину?
13. Яка максимальна довжина рядка в текстовому файлі?
14. Чи можна текстовий файл відкрити одночасно для читання та запису?

Завдання для самостійного розв'язання

1. Створіть файл **`input.txt`**, який містить ваше прізвище, ім'я та по батькові. Відкрийте його для читання та виведіть інформацію на екран. Необхідно розглянути дві ситуації: коли файл знаходиться разом з програмою (тобто не треба вказувати шлях до файлу) і коли файл знаходиться в будь-якій іншій директорії. Крім того, слід передбачити повідомлення у разі, якщо файл не може бути відкритий.

2. Створіть три файли: **`input1.txt`**, **`input2.txt`** та **`input3.txt`**, в які занесіть відповідно ваше прізвище, ім'я та по батькові. Відкрийте файл output.txt для додавання та запишіть до нього в три рядки інформацію з трьох вхідних файлів.

3. Створіть бінарний файл **`bin.dat`** і запишіть до нього 16 випадкових цілих чисел, що складаються з двох цифр. Виведіть на екран вміст цього файлу, а також виведіть число, яке знаходиться у цьому файлі на тому місці, яке відповідає номеру вашого варіанта.

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. 1 / уклад. : Порошин С. М., Статкус А. В., Носик А. М., Онищенко В. В. – Харків : НТУ «ХПІ», 2021. – 158 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 1 / уклад. : Онищенко В. В., Статкус А. В., Носик А. М., Корольова Я. Ю. – Харків : НТУ «ХПІ», 2021. – 204 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 1

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. 30.

Підписано до друку 02.04 .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