Міністерство освіти і науки України
Житомирський державний технологічний університет

В.Ю. Вінник

Алгоритмічні мови та основи програмування: мова C

Навчальний посібник

*Друкується за рішенням Вченої Ради
Житомирського державного
технологічного університету
(протокол № 4 від 03.12.2007)*

Житомир
2007

УДК 004.423.2 (075)
ББК 32.973-018.1я75
В 48

Вінник В.Ю. Алгоритмічні мови та основи програмування: мова С. — Житомир: ЖДТУ, 2007. — 328 с.
ISBN 978–966–683–143–2

Викладено фрагмент мови С, достатньо потужний, щоб розв'язувати реальні задачі, та водночас не переобтяжений другорядними подробицями. Роз'яснено найважливіші поняття програмування, основні техніки та прийоми програмування у структурно-модульному стилі. Виклад супроводжується розбором великої кількості прикладів та задачами для самостійної підготовки.
Посібник призначений для студентів технічних спеціальностей, які починають вивчати програмування, а також для всіх, хто вивчає програмування самостійно.

Рецензенти:
академік НАН України В.Н.Редько,
д.т.н., професор А.В. Панішев

ISBN 978–966–683–143–2 © В.Ю. Вінник, 2007

Зміст

Вступ **7**

1. Перше знайомство з програмуванням **8**
1.1. Загальні поняття . . . 8
1.2. Змінні, значення та типи . . . 10
1.3. Блок-схеми . . . 12
1.4. Покрокове виконання алгоритмів . . . 14
1.5. Двійкова система числення . . . 15
1.6. Двійкова арифметика: додавання . . . 17
1.7. Підсумковий огляд . . . 19

2. Основи мови Сі **20**
2.1. Приклад простої програми . . . 20
2.2. Ідентифікатори, імена змінних . . . 23
2.3. Числові константи та оголошення змінних . . . 24
2.4. Вирази, присвоювання . . . 25
2.5. Приклад . . . 27
2.6. Найпростіші засоби виведення . . . 28
2.7. Форматоване виведення чисел . . . 30
2.8. Функція введення . . . 31
2.9. Підсумковий огляд . . . 33

3. Арифметичні та логічні операції. Структури управління **34**
3.1. Арифметичні операції та перетворення типів . . . 34
3.2. Логічні операції та операції порівняння . . . 36
3.3. Умовний оператор та розгалужені алгоритми . . . 37
3.4. Оператори циклу . . . 40
3.5. Додаткові можливості циклів . . . 41
3.6. Оператор циклу з параметром . . . 44
3.7. Оператор вибору . . . 45
3.8. Підсумковий огляд . . . 47

4. Розбір прикладів на застосування структурних операторів **49**
4.1. Умовний оператор з простою умовою . . . 49
4.2. Складна умова . . . 50
4.3. Вкладені умовні оператори . . . 50
4.4. Взаємодія кількох умовних операторів . . . 51
4.5. Цикл з параметром . . . 52
4.6. Цикл з умовним оператором в тілі . . . 53
4.7. Вкладений цикл . . . 54
4.8. Складні вкладені цикли . . . 55
4.9. Максимум в послідовності . . . 56
4.10. Сума числового ряду . . . 57
4.11. Рекурентна послідовність . . . 59

5. Функції **61**
5.1. Поняття функції . . . 61
5.2. Важливі додаткові відомості . . . 63
5.3. Локальні та глобальні змінні . . . 65
5.4. Виклик та передача значень аргументів . . . 67
5.5. Розбір прикладу виклику функції . . . 69
5.6. Рекурсія . . . 71
5.7. Підсумковий огляд . . . 73

6. Масиви. Макропідстановки **74**
6.1. Макропідстановки . . . 74
6.2. Поняття масиву . . . 76
6.3. Типова схема обробки масивів . . . 78
6.4. Обчислення значень поліному . . . 79
6.5. Лінійний пошук . . . 81
6.6. Сортування: метод бульбашки . . . 82
6.7. Пошук поділом навпіл . . . 84
6.8. Прості числа, решето Ератосфена . . . 85
6.9. Підсумковий огляд . . . 88

7. Покажчики та динамічна пам'ять **89**
7.1. Основні поняття . . . 89
7.2. Особливості покажчиків . . . 90
7.3. Покажчики як аргументи та значення функцій . . . 92
7.4. Покажчики та масиви . . . 94
7.5. Динамічна пам'ять . . . 96
7.6. Особливості динамічно створених даних . . . 98
7.7. Покажчики вищих рівнів . . . 99
7.8. Багатовимірні масиви . . . 100
7.9. Підсумковий огляд . . . 101

8. Обробка текстових рядків **103**
8.1. Символьний тип . . . 103
8.2. Функції класифікації. Порядок символів . . . 104
8.3. Зберігання рядків в пам'яті . . . 105
8.4. Введення-виведення рядків . . . 106
8.5. Функції для обробки рядків . . . 107
8.6. Рядки в динамічій пам'яті . . . 109
8.7. Підрахунок числа пробілів . . . 110
8.8. Зсув та вставка . . . 111
8.9. Підрахунок числа слів . . . 113
8.10. Пошук підслова . . . 114
8.11. Підсумковий огляд . . . 116

9. Структурні типи **117**
9.1. Основні поняття . . . 117
9.2. Додаткові можливості . . . 118
9.3. Масиви структур . . . 121
9.4. Покажчики на структури . . . 122
9.5. Структури і функції . . . 124
9.6. Підсумковий огляд . . . 126

10. Обробка файлів, частина 1 **127**
10.1. Основні поняття . . . 127
10.2. Відкриття та закриття файлу . . . 128
10.3. Введення-виведення тексту . . . 130
10.4. Простий приклад . . . 133
10.5. Посимвольна обробка . . . 134
10.6. Посимвольна обробка: продовження . . . 136
10.7. Пристрої як спеціальні файли . . . 139

11. Обробка файлів, частина 2 **141**
11.1. Двійкові файли . . . 141
11.2. Прості приклади . . . 142
11.3. Файли структур . . . 144
11.4. Переміщення по файлу . . . 146
11.5. Способи організації файлів . . . 148
11.6. Підсумковий огляд . . . 151

12. Модульність **152**
12.1. Основні ідеї . . . 152
12.2. Включення файлів . . . 153
12.3. Проблема подвійного включення . . . 155
12.4. Умовна компіляція . . . 156

А. Поглиблений опис деяких елементів мови **158**
А.1. Форматні рядки функції printf . . . 158
А.2. Присвоювання . . . 160
А.3. Побітові операції . . . 161
А.4. Умовна операція . . . 161
А.5. Оператор переходу . . . 162
А.6. Покажчики на функції . . . 162
А.7. Статичні змінні у функціях . . . 164

Б. Небезпечні помилки **165**
Б.1. Ділення цілих та дійсних чисел . . . 165
Б.2. Пропущений знак = в логічному виразі . . . 166
Б.3. Узгодженість специфікаторів формату . . . 166
Б.4. Відсутність & в аргументі функції scanf . . . 167
Б.5. Числові значення символів-цифр . . . 167
Б.6. Символьні константи та однолітерні рядки . . . 168
Б.7. Порівняння масивів та адрес . . . 168
Б.8. Присвоювання масивів та адрес . . . 169
Б.9. Крапка з комою в структурному операторі . . . 170
Б.10. Складні умови . . . 171
Б.11. Логічна та побітова кон'юнкція і диз'юнкція . . . 171
Б.12. Помилки при роботі з пам'яттю . . . 172

В. Поради щодо стилю **175**
В.1. Імена змінних та функцій . . . 175
В.2. Оформлення виразів . . . 176
В.3. Відступи . . . 177
В.4. Оформлення тексту в цілому . . . 178
В.5. Стиль означень функцій . . . 179

Г. Словник термінів **181**

Д. Найважливіші слова мови Сі **186**

Бібліоґрафія **192**

Вступ

Даний навчальний курс є вступом до величезної галузі знань, яка продовжує свій бурхливий розвиток, — програмування. Для студентів цей посібник має стати першим знайомством з майбутньою спеціальністю.

Найважливіші поняття та практичні прийоми програмування вивчаються з залученням мови C, яка відіграла в історії програмування надзвичайно важливу роль та зберігає цю роль і сьогодні. Мовою C створено величезний обсяг системного та прикладного програмного забезпечення. Достатньо сказати, що мовою C написана операційна система UNIX (а також клон системи UNIX, ОС Linux, популярність якої стрімко зростає). Крім того, на мові C базується об'єктно-орієнтована мова C++, без знання якої неможливо уявити професіонала-програміста.

Методичні та дидактичні чинники обумовлюють особливий підхід до викладу мови програмування для початківців, тим більше такої складної, як мова C. Основна задача ознайомчого курсу основ програмування полягає не стільки в тому, щоб навчити студента самій лише мові C, скільки в тому, щоб взагалі навчити програмуванню і притаманному йому стилю мислення. Для порівняння, якщо досвідченому програмісту, щоб навчити новій мові, достатньо сказати, які в ній є типи даних, як пишеться оператор циклу, як оголошуються підпрограми, то початківцю треба спершу пояснити, що таке взагалі тип даних, оператор циклу та підпрограма.

Мова C має надзвичайно багатий спектр можливостей і доволі тонких засобів. Вивчення їх у повному обсязі на першому курсі майже неможливе та водночас і непотрібне. Тут викладено підмножину мови C, яка складається з найнеобхідніших на практиці засобів. Автор навмисно відкидає безліч елементів мови C, які самі по собі цікаві та важливі для реального практичного програмування, але в контексті першого курсу лише засмічували б виклад зайвими подробицями. Більш глибоке занурення в тонкощі досягається лише досвідом самостійної роботи та поступово здійснюється на старших курсах, коли вивчаються такі дисципліни, як системне програмування та сучасні технології програмування.

В цьому посібнику часто застосовується двоступеневий підхід до опису складних понять та термінів: при першому розгляді поняття пояснюється з грубим спрощенням, аби лише воно стало зрозумілим, а в наступних розділах вводиться більш строге пояснення. Це пов'язане зі специфікою мови програмування як предмету вивчення: найнеобхідніших понять і перехресних зв'язків між ними так багато, що ці поняття майже неможливо вишикувати в лінійну логічну послідовність: розповідаючи про одне поняття мови програмування, потрібно так чи інакше залучати до розгляду багато інших.

Іноді жертвуючи подробицями, автор намагається натомість сформувати у студента цілісну картину предмету. Тому в посібнику прийнято такий підхід, коли пояснюється не одне відірване від контексту поняття, а одразу система взаємопов'язаних понять. При цьому розгляд з необхідністю стає ітеративним: спочатку окреслюється загальна панорама предмету, а потім цей же матеріал повторюється з більшою деталізацією.

Побудова посібника повторює послідовність викладу матеріалу в лекціях. Кожен розділ супроводжується набором вправ. Їх вчасне виконання (одразу після відповідної лекції) гарантує якісне засвоєння матеріалу і, як наслідок, успішну здачу іспиту.

Автор вдячний Н.Толстіхіній за кропітку роботу з вичитки тексту.

Розділ 1

Перше знайомство з програмуванням

1.1. Загальні поняття

Ключові слова: мова програмування високого та низького рівня, транслятор, трансляція, алгоритм

Комп'ютер являє собою пристрій, який може виконувати задану послідовність дій. Діями можуть бути не лише обчислення, але також і відображення інформації на екрані, керування різноманітними зовнішніми пристроями тощо.

Програма являє собою опис тієї послідовності дій, яку повинен виконати комп'ютер. Цей опис з точки зору машини виглядає як послідовність *машинних команд* — команд, які машина вміє виконувати, оскільки механізми їх виконання реалізовано в електронній схемі комп'ютера. Сукупність машинних команд складає так звану *машинну мову*, або *мову низького рівня.* В епоху становлення обчислювальної техніки програмісти створювали програми саме в машинній мові.

Однак машинна мова є зручною та «рідною» саме для машини і зовсім незручна для людини. Працюючи над програмою в машинних кодах, програміст змушений насильно підлаштовувати своє мислення під технічні особливості машини. Внаслідок цього продуктивність праці програміста при роботі в машинній мові виявляється дуже низькою, особливо при розробці великих за обсягом програм.

Очевидно, вигідніше машину підлаштовувати під людські потреби та уявлення про зручність. Для цього призначені *мови програмування високого рівня*, які являють собою посередника, проміжний прошарок[1] між людиною з притаманним їй способом мислення та машиною, яка розуміє лише машинні команди. З одного боку, мова високого рівня розробляється так, щоб текст програми ясно відображав її сенс та задум. З іншого боку, мова високого рівня проектується таким чином, щоб будь-який текст можна було б перекласти з неї на машинну мову. Такий переклад робить не людина, а спеціальна програма, яка називається *транслятором* (рис. 1.1).

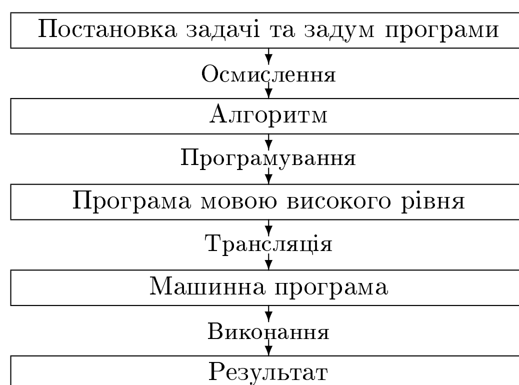

Рис. 1.1. Схема процесу створення та використання програми

Відомі два основні різновиди трансляторів: компілятори та інтерпретатори. *Компілятори* спочатку повністю перекладають весь текст програми з мови високого рівня мовою машинних команд, щоб потім можна було запускати отриману в результаті цього машинну програму. При використанні компілятору етап трансляції програми чітко відділено від етапу її

[1] Або, застосовуючи широко розповсюджений в програмістському середовищі термін, *інтерфейс.*

виконання: достатньо один раз відкомпілювати програму, щоб потім запускати отриманий результат безліч разів. Фактично, після компіляції вихідний текст програми, який був написаний мовою високого рівня, більше не потрібен. *Інтерпретатори* натомість читають текст пограми мовою високого рівня та виконують його по мірі прочитання. Переклад програми на машинну мову не запам'ятовується, отже, щоб виконати ту ж програму вдруге, її потрібно знов пропустити через інтерпретатор.

Компілятори з одного боку та інтерпретатори з іншого мають свої недоліки та переваги. Навіть якщо текст програми недописаний чи містить деякі помилки, інтерпретатор може виконати хоча б частину її тексту, поки не натрапить на першу помилку. Компілятор мусить спочатку перекласти машинною мовою весь текст програми, і лише після цього її можна запустити на виконання. Якщо програма містить хоча б одну помилку, компілятор не створить машинної програми, отже зробити пробний запуск такої програми неможливо. Таким чином, інтерпретатори набагато більш гнучкі. Але оборотною стороною є ненадійність інтерпретаторів: компілятор принаймні гарантує, що програму можна буде запустити на виконання лише тоді, коли в усьому її тексті немає граматичних помилок, а коли програма виконується під інтерпретатором, в ній можуть міститися «міни уповільненої дії» — помилки, які проявлять себе лише тоді, коли до них дійде виконання.

Крім того, запуск відкомпільованої програми вимагає набагато менших машинних ресурсів (пам'яті та процесорного часу): компілятор один раз і наперед виконує всю складну роботу з розбору, аналізу вихідного тексту програми та підготовки машинного коду, тому в подальшому запускається програма, вже перетворена у найзручнішу для машинного виконання форму. Якщо ж одну й ту саму програму кілька разів запускати під інтерпретатором, то інтерпретатор щоразу буде змушений заново читати та аналізувати її текст.

Існує велика кількість різноманітних мов високого рівня, більшість з яких орієнтовані на певні галузі застосувань або на обмежені класи задач. Це означає, що мову створюють таким чином, щоб задачі з даної галузі програмувалися в цій мові якнайпростіше. Зворотній бік спеціалізованих мов полягає в тому, що задачі з інших галузей програмувати в них дуже не зручно. Деякі мови, навпаки, є (відносно) універсальними — автори цих мов намагалися, щоб такою мовою можна було запрограмувати якнайширше коло задач з різних галузей. В цьому посібнику вивчається мова C, одна з найрозповсюдженіших на сьогоднішній день універсальних мов.

Вивчення будь-якої мови програмування складається з трьох основних рівнів. Перший (найнижчий) рівень вимагає вивчити правила запису текстів програм у даній мові — її словник (список *ключових слів*), розділові знаки та правила поєднання слів і знаків між собою. Цей рівень називають *синтаксисом.* На другому рівні стоїть знання точного значення слів та конструкцій мови: які дії комп'ютера вони означають (*семантика*). Вищий рівень (*прагматика*) вивчає, для яких задач та як саме слід застосовувати ці мовні конструкції. В процесі навчання ці рівні розгляду мови неможливо відокремити один від одного, їх слід вивчати лише у взаємозв'язку.

Отже, програма являє собою опис способу вирішення певної задачі, записаний у мові з жорстко заданими, формалізованими правилами. Жорстка заданість правил мови становить важливу та характерну відмінність програмування від інших, відомих зі шкільного курсу, способів запису методів розв'язання задач, і, відповідно, відмінність штучно створених мов програмування від людських (природних) мов.

Справді, людській мові притаманна багатозначність, образність, яскравість, метафоричність, оборотною стороною яких є деяка нечіткість, іноді нелогічність. Щоб зрозуміти текст людською мовою, потрібно мати уяву та інтуїцію, притаманну лише людині. Але машина не має інтуїції та уяви і може виконувати лише в точності ті команди, які записані в тексті програми. Машина виступає як «ідеальний бюрократ», який лише слідує інструкціям, не замислюючись над їх задумом (іншими словами, машина не вміє читати думки програміста).

Плануючи шляхи розв'язання задачі, потрібно мати на увазі, що машина не може інтуїтивно зрозуміти деякі звороти, які людині здаються цілком очевидними. Наприклад, будь-який студент зрозуміє таке означення: «факторіалом числа n є добуток $1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$», тоді як машина не може зрозуміти звороту «і так далі», позначеного у формулі трикрапкою.

Натомість, для того, щоб пояснити машині, як обчислювати факторіал заданого числа n (тобто написати програму обчислення факторіалу), потрібно написати детальну інструкцію, як перебирати числа від 1 до n.

Увага! Написання програми потребує абсолютно чіткого уявлення про те, яким чином повинна розв'язуватися задача: з яких елементарних кроків (простих дій) та у якій послідовності складається процес розв'язання задачі.

Перш ніж писати програму в конкретній мові програмування, потрібно накреслити в своїй уяві логіку задачі. Зрештою, хоча знання мови програмування абсолютно необхідне для того, щоб бути програмістом, вміння мислити чітко і логічно ще важливіше.

Спосіб вирішення задачі, розкладений у логічну послідовність елементарних дій та зв'язків між ними, називається *алгоритмом*. Зрозумівши алгоритм, його вже неважко записати у будь-якій відомій мові програмування. Отже, текст програми є лише *формою втілення* деякого алгоритму, а алгоритм є *змістом*, або *сенсом* тексту програми.

Увага! Алгоритм — це сукупність елементарних операцій та правила, які визначають, в якому порядку ці операції виконуються.

В загальному підсумку маємо, що процес розробки та використання програми можна зобразити схемою рис. 1.1.

Запитання

1. Що таке мова програмування високого рівня? низького рівня?
2. Що таке транслятор?
3. Що таке алгоритм?
4. Чи є алгоритмами (тобто, чи достатньо чітко описані для розуміння «ідеальним бюрократом» — обчислювальною машиною) методи розв'язку задач, відомі зі шкільного курсу математики:
   - Спосіб додавання чисел «у стовпчик»;
   - Метод Евкліда для знаходження найбільшого спільного дільника двох цілих чисел;
   - Спосіб знаходження гіпотенузи прямокутного трикутника за двома відомими катетами;
   - Спосіб розв'язання квадратного рівняння;
5. Опишіть алгоритм поїздки з дому до навчання (припустимо, що елементарними є такі дії, як «відчинити двері», «пройти від тролейбусної зупинки до університету» тощо).

1.2. Змінні, значення та типи

Ключові слова: змінна, значення змінної, поточний стан пам'яті, вираз, присвоювання, оператор, тип даних

Програма здійснює свої обчислення над деякими величинами, або даними. Наприклад, програма знаходження середнього арифметичного з двох чисел обробляє величини a, b (вихідні дані) та m (результат).

В програмуванні, як і в математиці, прийнято такі величини називати *змінними*. Кожна змінна в програмі повинна мати своє *ім'я* — в даному прикладі іменами змінних є «a», «b» та «m». Змінна в кожен момент часу має деяке *значення*, але значення кожної змінної в процесі роботи програми може змінюватися. Наприклад, програма обрахунку середнього арифметичного для того й призначена, щоб значенням змінної m наприкінці її роботи стало число $\frac{a+b}{2}$ (на початку роботи програми це значення було ще невідоме).

Сукупність значень всіх змінних в деякий момент виконання алгоритму (програми) разом утворює поточний *стан пам'яті* обчислювальної машини. Так, в попередньому прикладі можна розглянути стан пам'яті, поклавши значення $a = 2$, $b = 6$, або інший стан: $a = 28$, $b = 32$.

Якщо відомі значення змінних, то алгоритм може обчислювати значення різноманітних *виразів*, що складаються з імен змінних, числових констант та арифметичних операцій. Наприклад, якщо дано значення змінних $a = 28$, $b = 32$, то результатом обчислення виразу $a+b$ стане значення 60, а значенням виразу $\frac{a+b}{2}$ — число 30.

В процесі роботи алгоритму змінній може *присвоюватися* нове значення, отримане в результаті обчислення певного виразу. Присвоювання будемо позначати як «$x \leftarrow E$», де x — ім'я змінної, E — вираз. Виконати присвоювання означає обчислити в даному стані пам'яті (тобто при даних значеннях змінних) значення виразу E та надалі вважати його значенням змінної x. Результатом присвоювання є новий стан пам'яті, в якому змінна x має нове значення, а інші змінні мають ті ж значення, що і в попередньому стані. Наприклад, якщо в стані $a = 28$, $b = 32$ виконати присвоювання $a \leftarrow a + b$, отримаємо новий стан $a = 60$, $b = 32$.

В той момент, коли змінній присвоюється значення, її старе значення безповоротно втрачається. Щоб підкреслити цю основну властивість, дану операцію називають ще *деструктивним*, або руйнівним присвоюванням.

Присвоювання є частковим випадком операторів. Під *операторами* розуміють дії, які може виконувати машина, та з яких складаються програми.

Увага! Згадайте означення алгоритму на с. 10: «алгоритм є сукупністю елементарних дій...». Для найрозповсюдженіших мов програмування саме присвоювання є тими самими елементарними діями. Присвоювання — «цеглинка» для побудови як завгодно складних програм.

В елементарній математиці значеннями змінних, як правило, бувають дійсні числа. Так, значенням змінної s з предметним змістом «площа» може бути число $2,4$. Але використовуються також змінні, значеннями яких за змістом задачі можуть бути лише цілі числа, як, наприклад, значення змінної i у формулі $M = \sum_{i=1}^{n} \frac{1}{i^2}$. В деяких формулах значеннями змінної можуть бути комплексні числа, матриці, вектори тощо.

При цьому з кожною змінною у математиці пов'язана не лише множина значень, які вона може приймати, але й сукупність операцій, які над цими змінними можна виконувати. Так, якщо змінні a та b дійсні числа, то допустимими є вирази $a + b$, $a - b$, $a \cdot b$, a/b, a^b. Але якщо змінні u та v мають своїми значеннями вектори, то вирази $u + v$, $u - v$, $u \cdot v$ допустимі, а вирази u/v та u^v не мають сенсу.

За допомогою таких міркувань приходимо до одного з центральних понять програмування — типу даних. *Типом* називається множина допустимих значень (об'єктів даних) разом з сукупністю операцій, які над цими значеннями можна виконувати. Крім того, тип визначає спосіб зберігання значення в машинній пам'яті, спосіб зображення значень двійковим кодом. Кожна змінна в алгоритмі або програмі має певний тип — це означає, що їй можуть присвоюватися лише ті значення, що належать даному типу, та над нею можуть виконуватися лише ті операції, що дозволені для даного типу.

В мовах програмування є декілька вбудованих, або стандартних типів, а також розвинені засоби, які дозволяють програмісту конструювати на власний розсуд нові, як завгодно складні похідні типи даних, доповнюючи ними мову. Поки що будемо розглядати лише два типи: цілий та дійсний.

Уявімо, наприклад, змінну, сенсом якої є кількість студентів у групі. Природно, що така змінна має цілий тип, оскільки в групі не може бути, скажімо, $20,67$ осіб. Водночас, змінна «середній бал студента по результатах зимової сесії» повинна мати дійсний тип.

Увага! Слід добре усвідомити важливу деталь: числа, такі як, наприклад, 12 та $12,0$ — це зовсім різні об'єкти. Перше з них цілого типу, а друге дробове — належить до дійсного типу. Попри те, що у другого числа дробова частина дорівнює нулю, і значить число $12,0$ за своїм значенням співпадає з числом 12, у числа $12,0$ *в принципі є дробова частина* (хоча й нульова), тоді як у числа 12 взагалі дробової частини немає. Ці числа зовсім по-різному зберігаються в пам'яті машини.

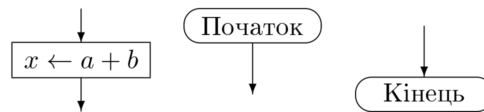

Рис. 1.2. Найпростіші позначення у блок-схемах

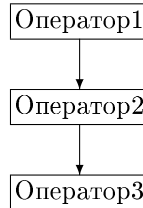

Рис. 1.3. Послідовне виконання операторів

Запитання

1. Що таке змінна, значення змінної, стан пам'яті?
2. Що таке тип даних?
3. Яка різниця між цілим числом та рівним йому за значенням дійсним числом?
4. Як використовуються змінні при роботі алгоритму? Що таке присвоювання?
5. Нехай дано стан пам'яті зі значеннями змінних $x = 12$, $y = 8$, $t = 1$. Який буде стан пам'яті після виконання оператору $t \leftarrow t + (x - y)$?

1.3. Блок-схеми

Ключові слова: блок-схема, передача управління, послідовне виконання, умовна конструкція, розгалуження, цикл, цикл з передумовою, цикл з постумовою

Згідно матеріалу попереднього розділу поняття алгоритму уточнюється таким чином: це сукупність операторів (зокрема, присвоювань) разом з сукупністю правил, які визначають, в якому порядку виконувати ці оператори. Виконання алгоритму починається в деякому початковому стані пам'яті та зі заздалегідь відомого початкового оператору і далі здійснюється крок за кроком. На кожному кроці виконується деякий оператор, який переводить машину в інший стан, та за деякими правилами визначається, який оператор буде виконуватися наступним. Виконання закінчується, коли алгоритм досягає оператору, заздалегідь визначеного як завершальний.

Зручним та наочним способом опису алгоритму є графічне зображення у вигляді блок-схеми. Кожен оператор зображується прямокутником; напис у прямокутнику вказує сенс оператору. Початковий та завершальний оператори мають спеціальні позначення, показані на рис.1.2.

В найпростішому випадку оператори з'єднані між собою стрілками таким чином, що з кожного оператору виходить рівно одна стрілка, яка йде на вхід наступного оператору. Це означає, що оператори утворюють ланцюжок і виконуються послідовно, один за одним. Так, у прикладі, показаному на рис. 1.3, спочатку виконується оператор1, потім оператор2, а за ним — оператор3.

Зрозуміло, що одного лише послідовного виконання недостатньо для вирішення всіх задач програмування. Справді, в послідовних програмах вся сукупність дій жорстко задана наперед, отже програма не може гнучко змінювати свою поведінку залежно від поточної ситуації. Таку гнучкість, або здатність приймати рішення «на ходу», абсолютно необхідну для всіх реальних задач, забезпечує *умовний* блок, який ще називають *розгалуженням* (рис. 1.4). Коли на умовний блок передається виконання (за стрілкою, що на рисунку входить зверху),

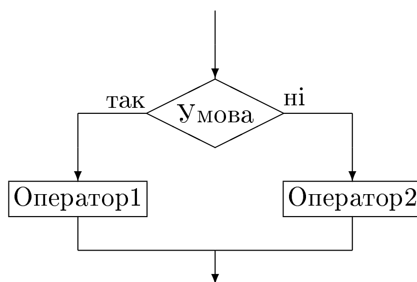

Рис. 1.4. Умовна конструкція (розгалуження)

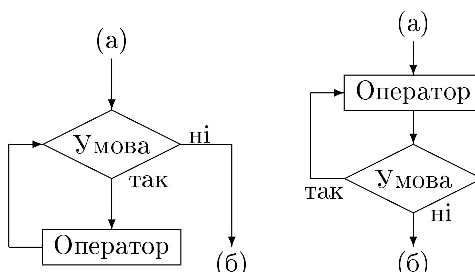

Рис. 1.5. Цикли з передумовою та постумовою

спочатку в поточному стані пам'яті перевіряється умова. Якщо умова істинна то виконується оператор1, в протилежному випадку виконуєтсья оператор2. Далі, незалежно від того, яким з двох шляхів пішло виконання умовного блоку, управління передаєтся на наступний блок, на який вказує стрілка, що на даному рисунку виходить вниз. Детальніше про умовну конструкцію та її застосування для конкретних задач мова буде йти в розділі 3.3.

Хоча умовна конструкція вже значно збагачує програмування, набагато розширюючи коло можливих програм, цих програмних структур ще зовсім не достатньо. Залишається ще один фундаментальний різновид програмних структур: повторювати деяку дію доти, поки залишається істинною деяка умова. Таку конструкцію називають *циклом* (рис. 1.5). Отже, циклічна конструкція забезпечує багаторазове виконання одного й того самого оператору (його називають *тілом циклу*), поки не буде досягнуто певної мети. Важливо, що кількість повторів заздалегідь невідома, вона визначається по ходу виконання алгоритму: тіло треба повторити не 5, 10 чи 100 разів, а стільки, скільки виявиться потрібним, щоб стала хибною умова.

Увага! На перший погляд може здатися, що ця схема нічим принципово не відрізняється від умовної структури: як в умовній конструкції, так і в циклі перевіряється умова, і залежно від неї виконання алгоритму продовжується одним з двох можливих шляхів. Але дуже важлива відмінність полягає в тому, що в умовній конструкції обидві гілки просувають виконання алгоритму ближче до завершення, тоді як в циклічній конструкцій одна з гілок направлена в бік завершення, а друга — у зворотній бік, повертаючи хід алгоритму до попередньої точки.

Розрізняють два основні різновиди циклу. Цикл з *передумовою* (на рисунку ліворуч) полягає в тому, що спочатку перевіряється умова, і, якщо вона істинна, виконується тіло циклу, після чого ця процедура повторюється знов. Неважко помітити, що такий цикл в загальному випадку забезпечує виконання тіла 0 чи більше разів: справді, якщо в момент входу в цикл за стрілкою (а) умова вже виявиться хибною, то управління одразу передається за стрілкою (б), обминаючи тіло. Цикл з *постумовою* (праворуч), натомість, спочатку виконує тіло, а потім вирішує, чи треба продовжувати далі. Такий цикл забезпечує виконання тіла 1 чи більше разів.

Насамкінець треба зазначити, що всього лише трьох цих структур управління, послідов-

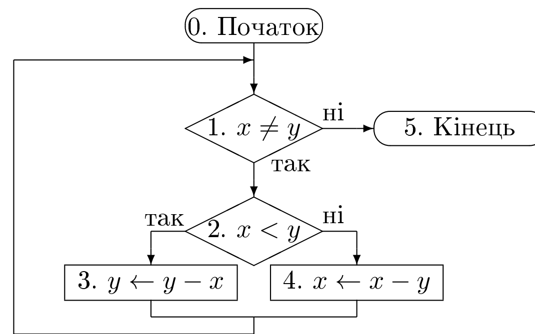

Рис. 1.6. Блок-схема алгоритму Евкліда

ної, умовної та циклічної, виявляється достатньо, щоб написати будь-яку програму. Різноманітні мови та методи програмування, хоча й містять складні та потужні засоби, залишаються основаними на цих трьох фундаментальних поняттях.

Запитання

1. Як зображуються на блок-схемі елементарний оператор (присвоювання), початок та кінець алгоритму?
2. Як позначаються та як виконуються послідовна і умовна конструкції?
3. Як позначаються та як виконуються циклічні конструкції? Що таке цикл з передумовою, з постумовою?

1.4. Покрокове виконання алгоритмів

Ключові слова: процес виконання алгоритму, крок виконання алгоритму

Розглянемо в деталях процес виконання алгоритму, заданого блок-схемою рис. 1.6. Алгоритм являє собою циклічну конструкцію з умовою в блоці 1, тілом якої є умовний оператор, у якого умова міститься в блоці 2, а гілками є оператори 3 та 4.

Нехай початковий стан пам'яті обчислювальної машини складається з двох змінних, значення яких відповідно $x = 15$, $y = 9$. Після запуску алгоритму управління передається на оператор розгалуження під номером 1. Перевірка умови $x \neq y$ при підстановці значень змінних з поточного стану пам'яті дає $15 \neq 9$ — істинне твердження. Оператор розгалуження лише перевіряє умову, не присвоюючи нових значень, тому стан пам'яті залишається незмінним: $x = 15$, $y = 9$. За стрілкою, поміченою словом «так», управління передається на оператор під номером 2, теж оператор розгалуження. Підстановка значень змінних дає співвідношення $(15 < 9)$, ця умова хибна, тому за стрілкою з позначкою «ні» управління передається на оператор 4, стан пам'яті поки що залишається незмінним.

Виконання оператору 4 полягає в тому, що обчислюється значення виразу при підстановці значень змінних: $x - y = 15 - 9 = 6$. Оператор присвоювання змінює стан пам'яті: значенням змінної x відтепер стає число 6, а значення змінної y залишається таким, як було раніше, тобто 9. За стрілкою приходимо знов до умовного оператору 1 — тіло циклу виконалося вже один раз, і тепер машина перевіряє, чи потрібно продовжувати виконання циклу.

Виконання оператору 1 в новому стані пам'яті встановлює, що умова $x \neq y$ істинна $(6 \neq 9)$, це призводить до продовження циклу. Відбувається перехід до оператору 2. Він, в свою чергу, передає управління на оператор 3, оскільки істинною виявляється умова $x < y$ $(6 < 9)$. Результатом виконання оператору 3 стає новий стан пам'яті, в якому $x = 6$ (значення не змінилося), $y = 3$ (нове значення). Управління передається на оператор 1.

Оператор 1 передає управління на оператор 2 (виконання циклу продовжується), оскільки твердження $x \neq y$ істинне. Оператор 2 з'ясовує, що співвідношення $x < y$ хибне та передає

Табл. 1.1. Протокол процесу виконання алгоритму Евкліда

Крок	Блок	x	y	Примітка
0	0	15	9	Початок. Перехід до 1
1	1	15	9	$x \neq y$ істинне, перехід до 2
2	2	15	9	$x < y$ хибне, перехід до 4
3	4	**6**	9	перехід до 1
4	1	6	9	$x \neq y$ істинне, перехід до 2
5	2	6	9	$x < y$ істинне, перехід до 3
6	3	6	**3**	перехід до 1
7	1	6	3	$x \neq y$ істинне, перехід до 2
8	2	6	3	$x < y$ хибне, перехід до 4
9	4	**3**	3	перехід до 1
10	1	3	3	$x \neq y$ хибне, перехід до 5
11	5	3	3	Алгоритм завершено

управління на оператор 4. Той, в свою чергу, присвоює змінній x нове значення, яке обчислюється з виразу $x - y$ при підстановці наявних в поточному стані значень змінних, це значення дорівнює 3. Управління знов передається на перевірку умови циклу, блок 1.

Цього разу умова $x \neq y$ виявляється хибною, і управління передається на оператор 5, який є завершальним. Отже, алгоритм завершує свою роботу, його результатом є прикінцевий стан пам'яті: $x = 3$, $y = 3$.

Весь процес виконання алгоритму зручно зобразити у вигляді таблиці. Перша колонка таблиці — номер кроку процесу виконання, друга — номер блоку, який на цьому кроці виконується. В наступних колонках наведено значення змінних, які складають стан пам'яті *після* виконання даного кроку. Якщо на даному кроці значення деякої змінної змінилося, домовимося нове значення позначати жирним шрифтом. В останню колонку будемо записувати істинність чи хибність умов в розгалуженнях та циклах, а також номер блоку, на який здійснюється перехід. Таким чином, описаному вище процесу застосування алгоритму до заданого початкового стану пам'яті відповідає табл. 1.1.

Щодо даного прикладу залишається сказати, що алгоритм, який тут розглядався, є алгоритмом Евкліда для обчислення найбільшого спільного дільника двох чисел.

Запитання

1. Застосувати алгоритм Евкліда до початкових даних: $x = 20$, $y = 28$. Протокол процесу застосування записати у вигляді таблиці. Переконатися, що обчислене алгоритмом число справді є найбільшим спільним дільником чисел 20 та 28.
2. Скласти блок-схему алгоритму, який порівнює значення змінних x та y та найбільше з них присвоює змінній m.

1.5. Двійкова система числення

Ключові слова: позиційна система числення, двійкова система числення, розряд, вага розряду.

Обчислювальна машина зберігає та обробляє дані у двійковій системі. Вільне володіння двійковою арифметикою, знання способу зображення даних у пам'яті надзвичайно важливе для всього подальшого навчання.

Щоб зручніше було описувати двійкову систему числення, варто почати з огляду звичної кожному десяткової системи. Розглянемо, як позначення числа будується з окремих цифр, і навпаки, як з запису числа у вигляді послідовності десяткових цифр слідує його значення.

В кожній позиції десяткового запису числа може стояти одна з цифр набору $0 \dots 9$. Вклад кожної цифри в величину числа визначаєтья не лише самою цією цифрою, але й тим, на якій позиції вона стоїть. Кожна позиція має свою *вагу*, вага крайньої правої позиції дорівнює 1, а вага кожної наступної позиції в 10 разів більша, ніж у попередньої. Візьмемо, наприклад, число 7068. Розшифровка цього запису така: число містить вісім одиниць, шість десятків, нуль сотень та сім тисяч:

$$7068_{10} = 7 \cdot 10^3 + 0 \cdot 10^2 + 6 \cdot 10^1 + 8 \cdot 10^0.$$

Взагалі, якщо дано деякий n-розрядний запис $a_{n-1} \dots a_1 a_0$, де a_i — десяткові цифри, то його значенням є число

$$m = a_{n-1} \cdot 10^{n-1} + \dots + a_1 \cdot 10^1 + a_0 \cdot 10^0.$$

Перейдемо від десяткової системи до загального випадку. В системі числення з основою k існують цифри від 0 до $k-1$, а вага кожної наступної позиції в k разів більша за попередню.

Тепер буде легко описати двійкову систему, якщо покласти $k = 2$. В цій системі числення є цифри 0 та 1. Вага кожного наступного розряду збільшується вдвічі. Візьмемо, наприклад, двійковий запис 11001011_2 та обчислимо його значення.

$$\begin{aligned} 11001011_2 = {} & \\ & = 1 \cdot 2^7 + 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + \\ & + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = \\ & = 128 + 64 + 8 + 2 + 1 = 203_{10}. \end{aligned}$$

Таким чином, для того, щоб перевести число з двійкової системи в десяткову, достатньо просто знайти, які розряди в двійковому записі мають значення 1, та додати відповідні степені двійки.

Увага! Програмісту варто постійно пам'ятати степені двійки: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536. Ці числа дуже часто трапляються в повсякденній практиці програмування.

Тепер розглянемо зворотний перехід, від запису числа у десятковій системі числення до двійкового запису. Нехай дано ціле число x, і його треба записати в двійковій системі числення. Розділимо його на 2 націло та знайдемо залишок від ділення, позначимо частку через x_1, а залишок[1] через a_0:

$$x : 2 = x_1 \mod a_0.$$

Число a_0 буде останнім (молодшим) розрядом двійкового запису числа x.

Далі таку ж процедуру повторюємо з числом x_1: ділимо його на 2 та знаходимо залишок від ділення:

$$x_1 : 2 = x_2 \mod a_1.$$

Продовжуючи ділити числа x_i націло на 2, отримуємо нові розряди a_i двійкового запису числа x. Цей процес рано чи пізно завершиться, коли після чергового кроку буде $x_n = 0$. Тоді

$$x = [a_n a_{n-1} \dots a_1 a_0]_2 .$$

Покажемо цей алгоритм на прикладі числа 1380 (рік Куликовської битви). Зручніше за все подати процес у вигляді таблиці (табл. 1.2). Записавши отримані від ділення залишки (останній стовпчик таблиці) в зворотному порядку, маємо:

$$1380_{10} = 10101100100_2.$$

Увага! Часто від початківців можна почути про «десяткові числа» та «двійкові числа». Це безграмотність та груба помилка! Немає окремих десяткових чи двійкових чисел, числа

[1]Залишок від ділення будь-якого числа на 2 може дорівнювати лише 0 або 1.

Табл. 1.2. Приклад переведення числа в двійкову систему числення

0.	1380: 2 =690	mod 0
1.	690: 2 =345	mod 0
2.	345: 2 =172	mod 1
3.	172: 2 =86	mod 0
4.	86: 2 =43	mod 0
5.	43: 2 =21	mod 1
6.	21: 2 =10	mod 1
7.	10: 2 =5	mod 0
8.	5: 2 =2	mod 1
9.	2: 2 =1	mod 0
10.	1: 2 =0	mod 1

одні й ті самі — є лише десяткова, двійкова (та багато інших) *форми запису* чисел, тобто системи числення.

Варто сказати кілька слів про позначення основи системи числення. Часто її позначають нижнім індексом після послідовності цифр, наприклад 1380_{10} — число, записане в десятковій системі, 101011_2 — в двійковій системі. Але зручніше буває позначати систему числення літерою: b для двійкової (від англ. binary), d для десяткової (від англ. decimal), наприклад 1380d, 101011b. Оскільки десяткова система є найрозповсюдженішою, часто її не позначають взагалі: якщо при деякій послідовності цифр немає жодного позначення системи числення, по замовчуванню вважаємо, що мається на увазі десяткова система.

Запитання

Переведіть у двійкову систему рік свого народження. Потім переведіть отримане число з двійкової системи у десяткову та переконайтеся, що зворотне переведення дало те ж саме число. Виконайте ту ж вправу з числом, зіставленим з останніх трьох цифр номеру телефону та з числом з чотирьох останніх цифр номеру паспорту.

1.6. Двійкова арифметика: додавання

Ключові слова: додавання в стовпчик, перенесення одиниці в старший розряд.

Розберемо правила, за якими виконуються найпростіші арифметичні дії над числами, записаними у двійковій системі числення.

Щоб додати два числа у двійковій системі числення, потрібно записати їх одне під одним, вирівнявши по молодшому розряду. Якщо числа мають різну кількість розрядів, до коротшого спереду (тобто до старших розрядів) потрібно дописати відповідну кількість нулів (очевидно, число від цього не змінюється).

Далі треба обробляти розряд за розрядом, починаючи з наймолодшого та рухаючись вліво, за таким правилом:

$$\begin{aligned} 0+0&=\ 0\\ 0+1&=\ 1\\ 1+0&=\ 1\\ 1+1&=10 \end{aligned}$$

В останньому випадку, при додаванні двох одиниць, відбувається перенос одиниці до старшого розряду.

Наприклад, додамо числа 157 та 44. Перш за все, переведемо їх у двійкову систему:

$$157\text{d} = 10011101\text{b},$$

$$44\text{d} = 101100\text{b}.$$

Друге число має на два розряди менше, ніж перше, тому доповнюємо його спереду двома нульовими розрядами. Запишемо отримані числа одне під одним порозрядно:

$$\begin{array}{ccccccccc} & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline & . & . & . & . & . & . & . & . \end{array}$$

Починаємо додавати молодші (праві) розряди. За правилом $1 + 0 = 1$, тому молодший розряд результату дорівнює 1, також $0 + 0 = 0$, тому другий справа розряд суми нульовий:

$$\begin{array}{ccccccccc} & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline & . & . & . & . & . & . & 0 & 1 \end{array}$$

В третьому справа розряді маємо $1 + 1 = 10$, тому в третій розряд результату йде 0, а 1 переноситься в наступний розряд:

$$\begin{array}{ccccccccc} & & & & & 1 & & & \\ & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline & . & . & . & . & . & 0 & 0 & 1 \end{array}$$

Тепер в четвертому розряді маємо вже з урахуванням перенесеної одиниці $1 + 1 + 1 = 10 + 1 = 11$. Отже, в четвертий розряд результату йде одиниця, а ще одна одиниця переноситься в наступний розряд:

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline & . & . & . & . & 1 & 0 & 0 & 1 \end{array}$$

У п'ятому розряді, враховуючи перенесену одиницю, $1 + 1 + 0 = 10$, тобто знову відбувається перенос:

$$\begin{array}{ccccccccc} & & & 1 & & & & & \\ & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline & . & . & . & 0 & 1 & 0 & 0 & 1 \end{array}$$

Після додавання $1 + 0 + 1 = 10$ отримуємо значення шостого розряду (0) та знов перенос одиниці в наступний розряд:

$$\begin{array}{ccccccccc} & & 1 & & & & & & \\ & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline & . & . & 0 & 0 & 1 & 0 & 0 & 1 \end{array}$$

Нарешті, в сьомому розряді перенесена одиниця додається до двох нулів, $1 + 0 + 0 = 1$, переносу в наступний розряд немає, тому восьмий розряд дорівнює $1 + 0 = 0$. Остаточно:

$$\begin{array}{ccccccccc} & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ + & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}$$

Переведемо отриманий результат у десяткову систему числення:

$$11001001\text{b} = 2^7 + 2^6 + 2^3 + 2^0 = 128 + 64 + 8 + 1 = 201.$$

Додавання в десятковій системі чисел 157 та 44 дає той же результат, отже додавання чисел у двійковій системі виконано правильно.

Запитання

Обчислити суми чисел у десятковій та двійковій системах та порівняти результати: $37+23$, $73+27$, $80+48$. Це означає: взяти два числа, обчислити їх суму (позначимо її через s), потім перевести ці числа у двійкову систему, виконати у двійковій системі операцію додавання, отриманий результат перевести в десяткову систему та порівняти з сумою s.

1.7. Підсумковий огляд

Мова низького рівня — це мова команд, безпосередньо зрозумілих для обчислювальної машини. Програмування мовою низького рівня зручне з точки зору машини, але незручне для людини. Для людини призначені мови високого рівня. Програма мовою високого рівня сама по собі не є зрозумілою для обчислювальної машини. Тому після того, як людина напише текст програми у мові високого рівня, він перекладається на мову низького рівня. Автоматичний переклад здійснює спеціальна програма — транслятор, який буває двох основних різновидів: компілятор та інтерпретатор.

Обчислювальна машина має пам'ять, яка складається з іменованих комірок — змінних. Змінна містить значення певного типу. Тип — це сукупність можливих значень плюс набір допустимих операцій над ними (з точки зору машини — також спосіб зберігання значення в пам'яті). Сукупність значень всіх змінних в даний момент часу утворює стан пам'яті.

Вираз складається з констант, імен змінних та знаків операцій. Значення виразів можуть обчислюватися в заданому стані пам'яті — при цьому замість імен змінніх підставляються відповідні значення. Оператор присвоювання змінює стан пам'яті, присвоюючи певній змінній значення, яке дає обчислення певного виразу у поточному стані пам'яті.

Алгоритм — це сукупність елементарних операцій плюс сукупність правил, які визначають послідовність їх виконання. В ролі елементарних операцій виступають присвоювання. Фундаментальними структурами управління послідовністю операцій є: послідовне виконання, розгалуження (умовна конструкція) та цикл.

Обчислювальна машина зберігає та обробляє числа, зображуючи їх у двійковій системі. В цій системі є лише дві цифри, 0 та 1. Значення кожного наступного розряду в двійковому записі числа вдвічі більше, ніж у попереднього розряду. Це дозволяє легко переводити числа з двійкової форми запису в десяткову. Для переведення з десяткової системи в двійкову треба розділити число на 2, тоді черговий розряд двійкового запису числа дорівнює залишку, а з часткою потрібно зробити таку саму дію. Спосіб додавання чисел, записаних в двійковій системі, принципово не відрізняється від відомого зі шкільного курсу алгоритму додавання «в стовпчик» чисел, записаних у десятковій системі — розряди обробляються у порядку від молодшого до старшого з перенесенням надлишку в наступний розряд.

Розділ 2

Основи мови С

2.1. Приклад простої програми

Ключові слова: директива підключення заголовочних файлів `include`, головна функція `main`, оголошення змінної, виклик функції, аргумент функції, коментар, функція виведення `printf`, функція введення `scanf`.

Перше знайомство з мовою С краще почати не з теоретичних відомостей, а з розбору конкретного прикладу.

```
/* Програма запитує  у користувача  значення кута
   у градусах, переводить кут у радіани та друкує
   значення косинусу
   (c) В.Ю.Вінник, 2012 */

/* підключення заголовків: */
#include<stdio.h> /* для введення-виведення */
#include<math.h> /* для математичних функцій */

/* головна функція програми */
int main() {
  /* оголошуються змінні */
  double alpha_deg, /* кут у градусах */
         alpha_rad; /* кут у радіанах */
  /* в наступному операторі нічого не вводиться!
     лише друкується повідомлення для користувача */
  printf( "Введіть␣значення␣кута␣в␣градусах\n" );
  /* число вводиться та заноситься у змінну */
  scanf( "%lf", &alpha_deg );
  /* обчислення за формулою */
  alpha_rad = alpha_deg * 3.1415926 / 180;
  /* друк значення змінної */
  printf( "%lf␣градусів␣=␣%lf␣радіан\n",
      alpha_deg, alpha_rad );
  /* друк значення виразу */
  printf( "косинус␣=␣%lf\n", cos( alpha_rad ) );
  return 0; /* так треба */
}
```

В цьому короткому прикладі вміщено майже всі основні елементи програми та засоби програмування, які будуть вивчатися в подальшому. Зараз прокоментуємо їх стисло, в подальших розділах цієї глави опишемо більш грунтовно, але в загальних рисах, а в пізніших главах пояснимо тонкі деталі.

Речення українською мовою (чи будь-які послідовності символів), вміщені в тексті програми між своєрідними дужками `/*...*/`, називаються *коментарями*. Коментарі можуть міститися в будь-якому місці програми. Якщо програма в цілому пишеться *для машини*, яка буде її виконувати, то коментарі пишуться *для людини*, яка буде цю програму читати та, можливо, змінювати. Транслятор ігнорує коментарі, тобто просто їх «не бачить».

В коментарі записують будь-які пояснення щодо того, яку задачу програма вирішує, як програма побудована і як працює. У грамотному, професійному програмуванні вважається «хорошим тоном» супроводжувати програму детальними коментарями. Справді, сучасне програмування є поставленою на потік, промисловою спільною діяльністю великих

колективів. Тому текст програми стосується не лише двох: програміста і його машини, а повинен бути зрозумілим для інших людей. Загальноприйнятим правилом є починати кожну програму або модуль з анотації, де вказано, що робить програма, хто та коли її написав.

Увага! Наявність коментарів в лабораторних роботах обов'язкова. Програми без коментарів розглядатися не будуть.

Два рядки вигляду `#include<...>` є *директивами включення заголовочних файлів*. Справжній сенс та принцип дії цих директив буде детально описано в розділі 12.2, а поки що обмежимося спрощеним поясненням. Підключати заголовочні файли потрібно для того, щоб в подальшому тексті програми можна було використовувати *стандартні функції*. В даному прикладі заголовочний файл `stdio.h` потрібен для використання стандартних функцій введення-виведення (`printf` — друк інформації на екран та `scanf` — введення з клавіатури). Другий заголовочний файл `math.h` робить доступними математичні функції, в тому числі функцію `cos`.

Кожен стандартний заголовочний файл являє собою набір *оголошень* деяких функцій, а кожна стандартна функція оголошується в певному заголовочному файлі. Для того, щоб використати в програмі деяку стандартну функцію, потрібно напочатку програми підключити відповідний заголовочний файл. Імена стандартних функцій та назви відповідних заголовочних файлів містяться в довідниках, найважливіші з них описані в цьому посібнику.

Вся основна частина програми (опис тих даних, над якими працює програма, і тих дій, які вона з цими даними виконує), оформлюється у вигляді так званої *головної функції*, яка завжди має ім'я `main` (англ. головний). В тексті програми вона має вигляд

```
int main() {
...
}
```

В наведеній програмі є лише одна функція, а саме головна. В подальшому побачимо, що програми, крім найпростіших, складаються з багатьох функцій. Але функція `main` є у будь-якій програмі. Якщо у програмі є декілька функцій, то виконання починається з функції `main`, а вона, в свою чергу, може викликати інші функції.

Щодо слова **`int`** в оголошенні головної функції та щодо оператору **`return`** `0` наприкінці поки що жодних пояснень не даємо. Студентам на даному етапі пропонується сприймати цю «заготовку» головної функції догматично, як даність. Належні пояснення будуть дані в розділі 5.2.

Як видно з прикладу, *тіло* функції береться у фігурні дужки `{` та `}`. Всі рядки, що стоять між цими дужками, об'єднуються між собою в одне велике ціле. Пізніше побачимо, що в мові C фігурними дужками позначається фактично будь-який фрагмент програмного тексту, який треба виокремити в самостійну одиницю.

Першим в тілі функції стоїть *оголошення змінних*. Згадавши (див. вище), що коментарі ігноруються транслятором, легко зрозуміти, що транслятор «бачить» цей фрагмент тексту програми в такому вигляді:

```
double alpha_deg, alpha_rad;
```

Цей рядок повідомляє транслятору, що далі у програмі будуть використовуватися змінні з іменами `alpha_deg` та `alpha_rad`, і що значеннями обох можуть бути дійсні (дробові) числа (назва **`double`** для дробового типу чисел означає подвійну точність їх зображення в машинній пам'яті). Іншими словами, даними, над якими працює програма, є дві числові величини, які позначаються такими іменами. Оброблюючи оголошення змінних, транслятор виділяє для їх зберігання пам'ять потрібного об'єму.

Наступний рядок

```
printf( "Введіть␣значення␣кута␣в␣градусах\n" );
```

містить *виклик* стандартної функції виведення `printf`. Іншими совами, функція `main` *звертається* до іншої функції, щоб та виконала для неї певну послугу. В найпростішому випадку функція `printf` просто друкує на екран текст. Символ `\n` означає переведення на новий

рядок: надрукувавши текст на екрані, машина переставить курсор на початок наступного рядку, отже те, що буде друкуватися далі, з'явиться на наступному рядку. Більш детальний опис функції `printf` та способів її використання буде дано в розділах 2.6 та 2.7.

Увага! Одна функція може звертатися до іншої функції (викликати її) — це один з основоположних принципів програмування у стилі C. Виклик виглядає так: ім'я функції і далі в круглих дужках *аргументи*, тобто ті значення, які передаються у функцію для обробки.

Наступним йде виклик функції введення `scanf`.

```
scanf( "%lf", &alpha_deg );
```

Читач може помітити, що, на відміну від попереднього рядка, тут виклик функції має два аргументи: перший — рядок `"\%lf"` та другий — вираз `&alpha_deg`.

Увага! Якщо у виклику функції кілька аргументів, вони розділяються комою.

Даний виклик призупиняє виконання програми, щоб користувач набрав на клавіатурі дані. Коли користувач закінчує введення, натиснувши клавішу Enter, функція `scanf` інтерпретує введені ним символи як запис дробового числа (тобто переводить введені символи в двійкове зрбраження числа) — на це вказує аргумент `"\%lf"`, а потім заносить отримане число у пам'ять, у змінну `alpha_deg`. Детальний опис функції `scanf` міститься в розділі 2.8.

```
alpha_rad = alpha_deg * 3.1415926 / 180;
```

Цей рядок виконує обчислення за формулою: значення змінної `alpha_deg` (а воно було щойно введене користувачем з клавіатури) множиться на число π та ділиться на 180. Отримане значення заноситься у змінну `alpha_rad`. Згідно формули, якщо значенням змінної `alpha_deg` було значення деякого кута, виміряного у градусах, то значенням змінної `alpha_rad` стане значення того ж кута, виміряне в радіанах.

```
printf( "%lf␣градусів␣=␣%lf␣радіан\n",
   alpha_deg, alpha_rad );
```

Тут виклик функції виведення має складніший вигляд, ніж в перший раз. Функція `printf` бере значення змінної `alpha_deg`, переводить його у десятковий запис (послідовність символів — цифр), підставляє у взятий в лапки текст замість першого *специфікатору* `%lf`, потім аналогічно бере числове значення змінної `alpha_rad`, переводить у символьне зображення і підставляє в текст замість другого специфікатору `\%lf`. Отриманий в результаті цих підстановок текст функція `printf` друкує на екран.

Нарешті, рядок

```
printf( "косинус␣=␣%lf", cos( alpha_rad ) );
```

відрізняється від попереднього тим, що спочатку виконується виклик (звертання) до функції `cos`, де в якості аргументу передається значення змінної `alpha_rad` (всі стандартні тригонометричні функції в мові C працюють лише над радіанними мірами кутів). Функція `cos` обчислює косинус кута і *повертає* отримане значення, яке, в свою чергу, стає аргументом функції `printf`. Функція `printf`, звичайно, друкує це значення разом з текстом «косинус =».

Увага! Виклик функції можна підставляти на місце аргументу при виклику іншої функції.

Отже, стисло розглянуто основні елементи програм у мові C. Далі потрібно описати їх більш детально.

Запитання

Для чого слугують і як позначаються найважливіші складові програми:

- директива підключення заголовочного файлу;
- коментар;
- головна функція;

- тіло функції;
- оголошення змінної;
- оператор присвоювання;
- функція виведення;
- функція введення;
- виклик функції.

2.2. Ідентифікатори, імена змінних

Ключові слова: ідентифікатор, ім'я змінної, ключове слово мови програмування

Змінні, функції, типи даних та деякі інші складові програм в мові C можуть позначатися не лише латинською літерою (як прийнято в математиці), але й послідовністю літер, цифр та знаків підкреслення «_», яка починається з літери[1].

Така послідовність латинських літер, цифр та знаків підкреслення «_», яка починається з літери або, можливо, зі знаку «_», називається *ідентифікатором*. Отже, імена змінних, функцій, типів тощо є частковими випадками ідентифікаторів.

Приклади правильних ідентифікаторів (зокрема, імен змінних): `x`, `M`, `cp23`, `cor2tef4w`, `EvenNumbersCount`, `warm_water_volume2`. Неправильні ідентифікатори: `2_top` (починається з цифри) `moon@light`, `step#process` (містять недопустимі символи).

Увага! В мові C розрізняються великі та відповідні малі літери: «`ask`» та «`Ask`» — це два зовсім різних ідентифікатори, між якими немає нічого спільного.

Мова C, взагалі кажучи, дозволяє програмісту називати змінні в програмі будь-яким чином на власний розсуд — головне, щоб ім'я задовольняло наведеному вище правилу. Але з міркувань зручності рекомендується обирати ім'я змінної так, щоб воно відображало сенс даної величини. Наприклад, змінну, предметним сенсом якої є кількість чого-небудь, варто назвати `count` (від англ. лічильник), змінну, яка відповідає куту нахилу рейки — `alpha` тощо. Змінні, предметним сенсом яких є номер якогось предмету у послідовності однотипних предметів, рекомендується називати `i`, `j`, `k`, як це зазвичай роблять у математиці.

Слід ще раз наголосити, що транслятору і машині «все одно», як програміст називає ту чи іншу змінну: сказане в попередньому абзаці є *рекомендація*, а не *правило*, його дотримання потрібно людині, а не машині.

Програмісти-початківці часто припускаються помилки, намагаючись зробити ідентифікатор з двох чи більше слів, розділених пробілами (наприклад, `warm water volume`). Помилку легко зрозуміти, оскільки пробіл не входить в число допустимих символів (див. означення ідентифікатору) та *розділяє* між собою ідентифікатори: в наведеному прикладі є не один ідентифікатор, а три.

Якщо те поняття з прикладної задачі, якому відповідає змінна, зручніше за все описати не одним, а кількома словами, можна зіставити ім'я змінної з відповідних слів, розділяючи їх підкресленнями (наприклад, змінну «кількість сторінок» можна назвати `pages_count`), а можна кожне слово починати з великої літери (наприклад, `WidePagesCount`). Перший спосіб частіше використовується при програмуванні в «класичному» стилі мови C, а другий частіше застосовується в більш новому стилі (хоча, звичайно ж, транслятору все одно).

Також набула поширення так звана угорська нотація, яка полягає в тому, щоб кожен ідентифікатор (зокрема, ім'я змінної) починати з префіксу, який розкриває тип та, можливо, інші особливості того, що цим ідентифікатором іменується. Наприклад, змінні цілого типу (тобто такі, предметним змістом яких є ціле число, див. наступний розділ) починають з літери `n` (від терміну «натуральне число»), змінні дійсного типу — з літери `f` тощо. Короткий опис угорської нотації міститься в додатку B.

---

[1] Строго кажучи, правила дозволяють починати імена і зі знаку підкреслення, але цього робити не рекомендується

Насамкінець треба підкреслити, що в мові C є певний перелік *ключових слів* — таких ідентифікаторів, значення та допустиме використання яких жорстко закріплене в правилах мови. Тому ім'я змінної в жодному разі не повинне співпадати з ключовим словом. Наприклад, не може бути в програмі змінної з іменем **do**: хоча це слово є цілком правильним ідентифікатором, але воно є в списку ключових слів і може використовуватися лише як оператор циклу. Список ключових слів можна знайти у довіднику.

Запитання

1. Які з цих ідентифікаторів правильні, а які неправильні, і чому?
   `great`, `mi%cro`, `vinnik8`, `10meters`, `my_age`, `якість`, `Stop It`, `ProperHeatingTime1`, $\Sigma\omega_1$.
2. Запропонуйте імена змінних, сенсом яких у задачі є: кількість автомобілів, висота, швидкість, об'єм бензину в баку, відстань до станції призначення, кут повороту платформи.
3. Чому ім'я змінної чи функції не може співпадати з ключовим словом мови?

2.3. Числові константи та оголошення змінних

Ключові слова: цілий тип **int**, дійсний тип **double**, константа, показникова форма запису дійсних чисел, оголошення змінної, ініціалізація змінної при її оголошенні.

В мові C є декілька вбудованих, або стандартних типів, а також розвинені засоби, які дозволяють програмісту конструювати на власний розсуд нові, як завгодно складні типи даних, доповнюючи ними мову. Типи в мові C мають імена, які є ідентифікаторами[1].

Зараз назвемо лише два найважливіших типи: цілий тип **int** (значеннями можуть бути додатні та від'ємні цілі числа, допускаються операції додавання, віднімання, множення, ділення націло, залишку від ділення та деякі інші) та дробовий з подвійною точністю **double** (значеннями є дробові числа з десятковою комою, операціями є додавання, віднімання, множення, ділення). Щодо цих типів є деякі тонкощі, які потребують окремого розгляду, але детальніші відомості про них відкладемо до розділу 3.1.

Константи цілого типу записуються в програмі звичайним чином, у вигляді послідовності десяткових цифр, перед якою може стояти знак мінус. Наприклад, в програмі можна використати цілі константи `0`, `1`, `144`, `-211` тощо[2].

Дробові числа, константи типу **double**, в найпростішому випадку складаються з цілої частини (послідовності цифр), крапки (в англомовних країнах замість коми в десяткових дробах використовується крапка) та дробової частини (ще однієї послідовності цифр). Перед дробовою константою може стояти знак мінус. Прикладами є константи `2.0`, `3.1415926`, `-2.41`.

Крім того, допускається записувати дробові числа в показниковій формі (відповідає математичному позначенню $a \cdot 10^k$): до мантиси, яка записується так, як описано вище, можна приписати літеру `e` та одразу за нею ціле число. Наприклад, константи `2.4e5`, `7.42e-4` та `-3.1e7` позначають відповідно числа $2,4 \cdot 10^5$, $7,42 \cdot 10^{-4}$ та $-3,1 \cdot 10^7$.

При читанні математичного тексту людині зазвичай зрозуміло або інтуїтивно, або за контекстом, в якій множині приймає значення та чи інша величина (наприклад, в одних задачах невідома x є цілим числом, в інших задачах — дійсним, комплексним числом, тензором тощо). В текстах програм, однак, є потреба явно вказувати тип кожної змінної. Це робиться за допомогою спеціальної конструкції, яка називається *оголошенням змінної*.

Оголошення потрібне для того, щоб транслятор міг заздалегідь знати, скільки машинної пам'яті потрібно зарезервувати для зберігання змінної, які над нею можна виконувати дії та яким саме чином. Загальне правило полягає в тому, що кожна змінна в програмі повинна бути

[1] Строго кажучи, допускаються і безіменні типи, які, однак, в даному курсі не розглядаються, щоб не «засмічувати» виклад зайвими подробицями.

[2] Строго кажучи, мова C дозволяє запис цілих констант не лише у десятковій системі числення, але й у шістнадцятковій та вісімковій, але ці питання виходять за рамки даного вводного курсу.

оголошена перед першим використанням. Всі оголошення в мові С повинні бути розташовані до операторів.

Оголошення в найпростішому випадку має одну з двох наступних форм:

```
тип ім'язмінної;
тип ім'язмінної, ..., ім'язмінної;
```

В першому випадку оголошується одна змінна з певним іменем певного типу, а в другому — одночасно оголошується кілька змінних з різними іменами одного й того самого типу. Оголошення завжди закінчується символом «крапка з комою» (часто початківці припускаються помилки, забуваючи поставити цей символ). Дуже корисна можливість — в при оголошенні змінної одразу ж надавати їй початкового значення. Таке оголошення називають ще оголошенням з *ініціалізацією*[1], воно позначається так:

```
тип ім'язмінної = константа;
```

Приклади оголошень:

```
int N;
double price, total_sum = 0;
int i, j, k = 1;
double alpha4 = 3.1415926;
```

Запитання

1. Як мовою С записати числові константи: $-13,85$, $5,37 \cdot 10^{-10}$, $-3,007 \cdot 10^{12}$?
2. Які з цих оголошень правильні, а які помилкові?
   - `int is_closed;`
   - `double alpha, beta, gamma, x;`
   - `int block1; block2;`
   - `int a, b, c`
   - `int brush_width, double thickness;`
3. Запишіть оголошення змінних, що відповідають таким величинам:
   - Температура води, об'єм води, маса води (дійсні);
   - Кількість всіх квартир в будинку, кількість однокімнатних, двокімнатних, трикімнатних квартир;
   - Опір першого резистора, опір другого резистора, напруга на першому резисторі, напруга на другому резисторі.

2.4. Вирази, присвоювання

Ключові слова: вираз, виклик функції, оператор присвоювання.

Вираз в мові С може складатися з числових констант, імен змінних, знаків арифметичних та інших операцій, круглих дужок. Поки що будемо розглядати лише операції додавання `+`, віднімання `-`, множення `*` та ділення `/`. Нижче наведено кілька прикладів виразів:

- `12`
- `56 + 40 * 2 - 38`
- `20.5 + 0.16 / (5.3 - 2.1)`

[1]Ініціалізація (від латинського «початковий») — розповсюджений в програмістському лексиконі термін з широким значенням, в загальному розумінні означає початкові опарації, підготовчі дії над будь-яким компонентом програмної системи

Перші два вирази мають цілий тип, оскільки в них входять лише значення цілого типу, останній вираз має дійсний тип. Припустимо, що в програмі оголошено змінні:

```
double price, total_sum, x;
int i, j, k;
```

Тоді наведені нижче приклади також є виразами:

- `(k + 18) * (i - 2 * j)`
- `i + price * x + k * total_sum * 1.05`
- `k`

Перший вираз має цілий тип, оскільки до нього входять лише цілі константи та цілі змінні. В другий вираз входять цілі змінні `i`, `k`, але оскільки сума цілого та дробового числа в загальному випадку є дробовим числом, то вираз в цілому має дійсний тип. Третій приклад ілюструє той факт, що ім'я змінної саме по собі є виразом.

До складу виразів можуть також входити виклики *функцій*. Детальний розгляд поняття функції, одного з найважливіших у мові С, відкладемо на подальші розділи, а поки що обмежимося кількома найпростішими прикладами. В мові С є стандартна функція обчислення синусу, яка має ім'я `sin` (нагадаємо, що імена функцій також є ідентифікаторами). Тоді якщо є деякий `вираз` дійсного типу, значенням якого є число α то `sin(вираз)` також є вираз дійсного типу, і його значенням є $\sin\alpha$ (кути вимірюються в радіанах). Подібним чином, в мові є функції `cos`, `exp` та інші, які безпосередньо відображають відповідні математичні функції. Функція `sqrt` відповідає видобуванню квадратного кореню: значенням виразу `sqrt(вираз)` є число $\sqrt{\alpha}$. Наприклад, виразом мови С є `12 * exp(x + sin(2 * x)) - sqrt(x + 10)`, він відповідає математичному виразу $12e^x + \sin 2x - \sqrt{x+10}$.

Оператор присвоювання в мові С в найпростішій формі має вигляд

```
ім'язмінної = вираз;
```

Знак «`=`» тут слід читати як «присвоїти», а не «дорівнює». Оператор присвоювання завжди закінчується символом «крапка з комою».

Робота оператору присвоювання полягає в тому, що спочатку обчислюється значення виразу, який стоїть справа від знаку «`=`», а потім отримане значення присвоюється змінній, ім'я якої стоїть в лівій частині.

В наступному фрагменті після необхідних оголошень наведено приклади операторів присвоювання:

```
double x, y, r, omega = 4, t = 0.327, phi0 = 1.15;
int i = 10;
x = -1;
y = r * sin( omega * t + phi0 );
i = i + 1;
```

В першому операторі змінній `x` просто присвоюється числове значення -1. В другому прикладі змінній `y` присвоюється значення, яке розраховується за формулою (формула дає для моменту часу t декартову координату y точки, яка обертається навколо початку координат з постійною кутовою швидкістю ω, з постійним радіусом r, з початковим кутом φ_0).

В третьому прикладі слід звернути увагу на те, що одна й та сама змінна `i` може входити і в ліву, і в праву частину оператору присвоювання. Виконання цього оператору полягає в тому, що береться значення змінної `i`, яке вона має в даний момент, до нього додається 1, після чого результат поміщується знов у змінну `i`. Отже, оператор збільшує значення змінної на 1.

Запитання

1. Нехай змінна `price` дійсного типу має сенс «ціна одного комп'ютера», а змінна `n` цілого типу має сенс «кількість комп'ютерів». Записати оголошення змінних та оператор, який присвоює змінній `total` дійсного типу значення загальної вартості цих комп'ютерів.

2. Нехай у змінній `R` зберігається значення опору (в омах), а у змінній `I` — значення струму (в амперах), що протікає через цей опір. Записати оголошення змінних та оператор присвоювання, що присвоює змінній `U` значення напруги (у вольтах) на даному опорі. Всі змінні дійсного типу.
3. Нехай у змінній `time_hour` зберігається величина проміжку часу в годинах. Записати оголошення та присвоїти змінній `time_min` значення цього ж проміжку в хвилинах.
4. Нехай у змінній `v0` зберігається значення початкової швидкості тіла, у змінній `a` прискорення, у змінній `t` — час (всі величини задано в одиницях Міжнародної системи). Записати оголошення та присвоїти змінній `s` значення шляху, що пройде тіло, рухаючись прямолінійно. Всі змінні дійсного типу.

2.5. Приклад

Застосуємо набуті знання до реальної задачі. Складемо програму, що розв'язує квадратне рівняння

$$x^2 + 3x - 4 = 0$$

Згадавши сказане в розділі 1.1, що важливо напочатку скласти власне уявлення про логічну структуру процесу розв'язування задачі, розглянемо процедуру «ручного» розв'язування даного рівняння. Для цього перетворимо його до загального вигляду

$$a \cdot x^2 + b \cdot x + c = 0$$

при $a = 1$, $b = 3$, $c = -4$.

Треба знайти дискримінант за формулою $D = b^2 - 4ac$, в даному прикладі обчислення дає $D = 9 + 4 \cdot 4 = 25$. Поки що не будемо звертати увагу на можливий випадок, коли $D < 0$, а будемо покладатися на те, що коефіцієнти рівняння підібрані вдало.

Тоді корені рівняння отримуємо за формулами

$$x_1 = \frac{-b + \sqrt{D}}{2a}, \qquad x_2 = \frac{-b - \sqrt{D}}{2a}.$$

Розрахунок дає $x_1 = 1$, $x_2 = -4$.

Для написання програми варто згадати (див. розділ 2.3), що кожну змінну перед першим використанням треба оголосити, і що всі оголошення в мові С повинні стояти до операторів. Отже, отримуємо основну частину програми (тобто внутрішню частину функції `main` — такі складові програми, як директиви підключення заголовків та виклик функцій введення-виведення поки що не розглядаємо):

```
double a, b, c;
double D;
double x_1, x_2;
a = 1;
b = 3;
c = -4;
D = b * b - 4 * a * c;
x_1 = (- b + sqrt( D )) / ( 2 * a );
x_2 = (- b - sqrt( D )) / ( 2 * a );
```

В перших трьох рядках оголошуються змінні, що відповідають коефіцієнтам рівняння, дискримінанту та кореням. Зазначимо, що транслятору все одно, оголошуються змінні в одному чи в різних рядках. В даному прикладі можна було б з тим же успіхом оголосити всі змінні в одному рядку, а можна було б кожну змінну оголосити окремо. Проте програмісти зазвичай намагаються розташовувати оголошення так, щоб текст програми в цілому виглядав естетично привабливішим та логічно структурованим. Тому в наведеному фрагменті

окремо оголошено вихідні дані, окремо проміжну змінну (дискримінант), а окремо кінцевий результат.

В наступних трьох рядках змінним, що відповідають коефіцієнтам рівняння, присвоюються початкові значення. Наступний рядок являє собою формулу для обчислення дискримінанту. Слід звернути увагу, що для піднесення в квадрат використовується множення змінної `b` на себе. Нарешті, в останніх двох рядках реалізовано формули для знаходження коренів рівняння.

Зазначимо, що наведена вище програма, хоча й правильна в суто математичному сенсі (обраховує правильний результат), далека від досконалості. Обрахунок квардатного кореню (як і більшість функцій дійсного аргументу зі стандартної бібліотеки) може вимагати багато процесорного часу. В даному прикладі квадратний корінь з дискримінанту обраховується двічі: в обох рядках, де присвоюються значення змінним `x_1` та `x_2`, міститься підвираз `sqrt(D)`. Значно економніше один раз заздалегідь обрахувати значення квадратного кореню (присвоївши його в допоміжну змінну, скажімо, `d`) і потім використати його в двох останніх операторах. Читачеві пропонується зробити це в якості вправи.

2.6. Найпростіші засоби виведення

Ключові слова: заголовочний файл `stdio.h`, функція `printf`, текстова константа, спеціальні символи.

Програма повинна не лише робити обчислення в пам'яті машини, але й взаємодіяти із зовнішнім світом: отримувати початкові дані для розрахунків та виводити результати. Справді, від самих по собі обчислень в пам'яті немає користі, оскільки результати обчислень у вигляді електричних імпульсів у мікросхемах невидимі для зовнішнього споживача, який потребує послуг програми.

Стандарт мови C містить широкий спектр функцій введення-виведення (далі в цьому розділі ВВ). В цьому розділі розглядаються засоби ВВ, з одного боку найпростіші, а з іншого — надзвичайно потужні та достатні для найрізноманітніших застосувань.

Базові функції ВВ оголошені в заголовочному файлі `stdio.h`, назва якого походить скорочено від англійського «standard input-output» — стандартне введення-виведення. Отже, якщо в програмі передбачається використовувати ВВ, на самому початку її тексту повинна стояти директива

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

Функція `printf` призначена для друку тексту на дисплей[1]. Її можливості надзвичайно широкі та різноманітні, їх повний опис зайняв би багато місця. Поки що почнемо з простих способів використання, а більш витончені будемо запроваджувати згодом по мірі потреби.

Дисплей у текстовому режимі працює як телетайп. По дисплею переміщується *курсор* (має вигляд маленької горизонтальної риски, що мигає), це друкувальна каретка, яка позначає місце, де буде надруковано наступний символ. При виведенні символу на дисплей символ відображається у тій позиції, де стояв курсор, а курсор переміщується на одну позицію праворуч. Якщо рухаючись праворуч, курсор досяг правої границі дисплею (кінець рядка), то він переходить на початок наступного рядка. Якщо курсор стоїть на останньому рядку, то вміст дисплею «прокручується» на один рядок вгору (верхній рядок при цьому втрачається), внизу дисплею один рядок звільняється, і курсор стає на його початок.

Для телетайпного друку на екран простого текстового повідомлення слід викликати функцію, передавши їй один аргумент — текст, взятий в подвійні лапки("), такий текст в лапках ще називають *текстовою константою*.

```
printf( "вітаю!" );
```

[1] Більш строго, для відправки тексту у потік виведення `stdout`, який, однак, частіше за все буває пов'язаний саме з дисплеєм.

Слід звернути увагу, що відкривальні та закривальні лапки не відрізняються, це один і той самий символ. Транслятор вважає, що перше входження символу " починає текстову константу, друге — її завершує, і так далі. В лапках може стояти будь-яка послідовність літер, цифр та інших символів, саме в такому вигляді вони будуть надруковані на дисплеї, винятком є лише деякі *спеціальні* послідовності символів, які зараз буде розглянуто.

Увага! Дуже часта помилка початківців — незакрита текстова константа. Не забувайте ставити лапки наприкінці текстової константи.

Нехай треба вивести на екран текст, в якому хоча б один раз міститься сам символ лапок ("). Наприклад, потрібно, щоб на екрані з'явився напис

```
твір "Слово о полку Ігоревім"
```

Якщо написати в програмі

```
printf( "твір␣"Слово о полку Ігоревім"" );
```

то транслятор зафіксує помилку. Справді, перший символ " починає текстову константу, другий символ " (одразу перед словом «Слово») її завершує, отже текст «о полку Ігоревім» взагалі опиняється за межами текстової константи. Тому потрібен якийсь особливий спосіб позначити, що ці лапки не закривають текстову константу, а є її частиною.

Автори мови C запровадили такий спосіб позначення: якщо лапки грають роль символу всередині текстової константи, потрібно писати комбінацію \". Отже, у програмі треба написати

```
printf( "твір␣\"Слово␣о␣полку␣Ігоревім\"" );
```

Увага! З точки зору машини послідовність \" виступає не як два символи, а як *один* символ лапок. Зокрема, в пам'яті він займе один байт. Два символи — це лише спосіб позначення, прийнятий в мові.

Таким самим чином позначається в текстовій константі символ одинарних лапок (апостроф) '.

Крім того, є декілька особливих «літер», які не зображуються на екрані у вигляді видимих знаків, натомість виведення на екран такого символу певним чином впливає на переміщення курсору.

Найважливішим з таких символів є переведення курсору на новий рядок, що позначається \n. Наприклад, виклик

```
printf( "Я\nвивчаю\nмову\nC" );
```

надрукує на екрані текст

```
Я
вивчаю
мову
C
```

Увага! Якщо останнім символом тексту, що друкується, є символ переведення рядку, то текст, який буде друкуватися *наступним* викликом функції `printf`, з'явиться на наступному рядку. Це гарний стиль.

Цікавий також символ \r — перехід на початок рядку. Друк цього символу призводить до того, що курсор повернеться на початок цього ж рядку, на якому він в даний момент стоїть, після чого наступні символи, які друкуються вслід за ним, будуть затирати (заміщувати) символи, які в рядку вже є. Розглянемо приклад:

```
printf( "добре" );
printf( "\rвідмінно" );
```

Виконання першого оператору призводить до того, що на екрані з'являється напис `добре`, а курсор стоїть одразу після слова. Одразу після цього виконується другий оператор, який перш за все переставляє курсор на початок рядка (текст все ще залишається незмінним).

Після цього нові літери друкуються поверх літер слова «добре», отже в результаті на дисплеї буде надруковано `відмінно`.

Увага! Ця можливість стає дуже корисною, коли треба під час роботи довгого алгоритму постійно виводити повідомлення про поточний стан його роботи. Програма виглядає гарно, якщо кожне нове інформаційне повідомлення замінює собою попереднє.

Ще один корисний символ — *табуляція*, яка у текстових константах позначається `\t`. Весь екран немов би розділений на вертикальні колонки однакової ширини у 8 символів. Друк символу табуляції `\t` призводить до того, що курсор переходить по горизонталі до найближчої колонки. Наприклад, фрагмент

```
printf( "Хліб\t2\tшт\n" );
printf( "Огірки\t1\tкг\n" );
printf( "Ск\t0,8\tл\n" );
```

надрукує текст

```
Хліб␣␣␣␣2␣␣␣␣␣␣␣шт
Огірки␣␣1␣␣␣␣␣␣␣кг
Ск␣␣␣␣␣␣0,8␣␣␣␣␣л
```

Читач за аналогією легко зможе самостійно отримати відповідь на

Запитання

Як в текстовій константі зобразити сам символ «\»? Написати фрагмент програми, що друкує на екран текст

```
==///<<<*!*>>>\\\==
```

2.7. Форматоване виведення чисел

Ключові слова: функція `printf`, форматний рядок, специфікатор формату

Основне багатство можливостей функції `printf` пов'язане з форматованим (тобто певним чином оформленим) друком різноманітних значень. В цьому випадку функції передається більша кількість аргументів[1].

Першим аргументом, як і в попередньому розділі (там це був взагалі єдиний аргумент), є текстова константа, яка називається *форматним рядком*. Крім звичайних символів, які будуть друкуватися на екран в незмінному вигляді, форматний рядок містить деяку (довільну) кількість так званих *специфікаторів формату* (див. нижче). Далі йдуть решта аргументів, їх кількість і типи повинні відповідати специфікаторам. Кожен специфікатор описує, в якому вигляді треба друкувати на екран відповідний аргумент.

В найпростішому випадку специфікатор складається з символу відсотку `%` та символу, який визначає спосіб переводу аргументу у текстовий вигляд. Для друку цілого (типу **`int`**) числа у десятковому вигляді слугує символ `d`, а для друку дійсного (типу **`double`**) числа — пара символів `lf`.

Приклад.

```
int k;
double m;
k = 12;
m = 1.05;
printf( "Вантаж␣номер␣%d,␣маса␣%lf␣кг", k, m );
```

[1] Зазвичай функція має певну, чітко визначену кількість аргументів, які їй необхідно передати при виклику. Але в той же час в мові C існує можливість робити функції зі змінною кількістю аргументів. Саме до таких належить функція `printf`.

Перший специфікатор `%d` пов'язаний з першим після форматного рядку аргументом — змінною `k`, а другий специфікатор `%lf` — зі змінною `m`. функція `printf` починає з того, що друкує на екран символи форматного рядку, поки не натрапить на перший символ `%`. Отже, буде надруковано символи `Вантаж номер` та пробіл на кінці.

Далі символ відсотка вказує, що наступні символи після нього призначені не для друку на екран, а для обробки аргументів. Обробляючи перший специфікатор, функція `printf` бере значення змінної `k` (завдяки оператору присвоювання воно дорівнює 12), переводить його в десятковий запис `12` і друкує на екран. Отже, на дисплеї в даний момент надруковано напис `Вантаж номер 12`.

Обробивши специфікатор формату, функція `printf` продовжує обробляти форматний рядок далі символ за символом. Всі звичайні символи, тобто такі, які не є частиною специфікаторів, знову друкуються на екран. Так друкується кома, пробіл, літери `м`, `а` і так далі, поки функція не дійде до наступного специфікатору.

Обробляючи другий специфікатор, функція `printf` візьме значення змінної `m` (рівне 1,05), надрукує його символьне зображення `1.05` і продовжить друкувати звичайні символи з форматного рядку. Отже, наведений фрагмент друкує напис

```
Вантаж номер 12, маса 1.05 кг
```

Ще більш гнучкі та потужні можливості форматованого виведення описано в додатку А.1.

Нагадаємо, що в якості аргументів функції можуть виступати не лише змінні, але й будь-які вирази, наприклад:

```
printf( "загальна␣вартість␣%lf␣грн", n * price );
```

Увага! Знак відсотку `%` у форматному рядку завжди позначає початок специфікатору формату. Тому якщо потрібно надрукувати сам знак `%`, то у форматному рядку треба писати `%%`, наприклад фрагмент

```
double x, m;
x = 37; /* кількість пального у баку */
m = 50; /* ємність баку */
printf( "Наповнення␣баку␣%lf%%", x * 100.0 / m );
```

друкує текст `Наповнення баку 74%`.

Запитання

1. Надрукувати значення змінної `x` дійсного типу, значення її квадрату та кубу з відповідним пояснювальним текстом, наприклад: `значення: 1.5, квадрат 2.25, куб 3.375`;
2. В програмі є дві змінні, `x` та `y` дійсного типу. Надрукувати, у скільки разів `x` більше за `y`.
3. Дописати виведення на екран коренів квадратного рівняння (див. основну частину програми на с. 27).

2.8. Функція введення

Ключові слова: функція `scanf`, амперсанд, успішне та неуспішне співставлення.

Для введення значень змінних призначена функція `scanf`. За один її виклик може вводитися значення однієї або декількох змінних.

Першим аргументом функції є форматний рядок, в якому є один або декілька специфікаторів. Після форматного рядка йде решта аргументів — *адреси змінних*, в які треба розмістити введені значення. Кількість змінних повинна в точності збігатися з кількістю специфікаторів, а їх типи повинні узгоджуватися зі специфікаторами.

Робота функції scanf полягає в тому, що робота програми призупиняється, і користувачу надається можливість набрати на клавіатурі[1] деяку послідовність символів. Коли користувач завершує введення, натиснувши клавішу Enter, функція scanf проводить розбір введеної послідовності символів, співставляючи її з форматним рядком. А саме, функція намагається проінтерпретувати введену послідовність у відповідності зі специфікатором та розміщує результат в задану змінну.

Наприклад, нехай в програмі оголошено змінну **int** x. Тоді для того, щоб ввести її значення, потрібно викликати функцію scanf у вигляді

```
scanf( "%d", &x );
```

Якщо користувач введе послідовність символів 123, то функція scanf у відповідності до специфікатору %d розгляне її як запис цілого числа у десятковій системі, отримає значення 123 і помістить його у змінну x.

Увага! Перед іменем кожної змінної у функції scanf обов'язково повинен стояти знак & (амперсанд)[2], його відсутність є помилкою, яку транслятор не помітить, і яка може непередбачуваним чином спотворити роботу програми.

Розглянемо тепер більш загальний випадок, коли у форматному рядку міститься деякий текст та кілька специфікаторів. Тоді функція scanf буде намагатися аналізувати символи, введені користувачем, доки це можливо, тобто доки ці символи відповідають форматному рядку. Коли функція scanf зустріне невідповідність між вводом користувача та форматом, якого вимагає форматний рядок, вона перериває свою роботу, в наслідок чого частина значень можуть залишитися не введеними.

Наприклад, розглянемо фрагмент

```
int k, l;
n = scanf( "A(%d!%d)", &k, &l );
```

Тоді функція scanf буде вимагати від користувача ввести таку послідовність символів, яка починається з літери A та відкриваьлної дужки, одразу після якої йде ціле число, потім знак оклику та друге ціле число та закривальна дужка. Якщо користувач введе A(23!46), що повністю відповідає формату, то функція успішно розбере цю послідовність символів та присвоїть змінним k та l відповідно значення 23 та 46.

Якщо ж ввести A(23,46) (замість знаку оклику, який вимагається форматним рядком, стоїть кома), то функція scanf успішно співставить початок форматного рядка "A(%d" з початком введеної послідовності A(23 та присвоїть змінній k значення 23. Але після цього вона зустріне у введеній послідовності символ коми, тоді як очікується знак оклику. Для введеної користувачем коми немає відповідності у форматному рядку, отже функція scanf перериває роботу, а значення змінної l залишається не введеним.

Нарешті, якщо користувач введе Q(23!46) або A[23!46], або Hello тощо, функція scanf взагалі не зможе співставити жодного введеного символу з форматним рядком та перериває роботу одразу ж, не ввівши значення жодної змінної.

Отже, функція стандартного введення в мові C дозволяє не лише вводити деякі числові значення, але й задавати вимоги до вигляду тексту, який вводить з клавіатури користувач. Як було щойно показано, функція може ввести змінних менше, ніж було замовлено. Тому необхідно вміти дізнатися в програмі, наскільки успішно відбулося введення.

Функція scanf повертає кількість успішно введених змінних. Це варто використовувати в своїх програмах для перевірки правильності даних, введених користувачем. Наприклад:

```
int k, l, n;
n = scanf( "A(%d!%d)", &k, &l );
printf( "%d␣змінних␣успішно␣введено\n", n );
```

[1] Строго кажучи, дана функція бере дані не з клавіатури, а зі стандартного потоку введення stdin, однак він частіше за все пов'язаний з клавіатурою.

[2] Що означає операція & і навіщо вона потрібна у функції введення, буде зрозуміло з подальшого, див. розділ 7.

Якщо користувач введе `A(23!46)`, то програма повідомить, що введено обидві змінні якщо введе `A(23,46)`, то програма відповість, що введено одну змінну, а якщо ввести `hello`, то відповіддю програми буде «`0 змінних успішно введено`».

Більш складні можливості функції `scanf`, що стосуються тонкої настройки формату введення, описані в довідниках.

Увага! Хорошим стилем програмування є перед введенням якихось даних друкувати на екран підказки для користувача, щоб той знав, чого програма від нього очікує. Наприклад:

```
printf( "Введіть␣кількість␣джерел␣струму:␣" );
scanf( "%d", &n_generators );
printf( "Введіть␣кут␣нахилу␣в␣градусах:␣" );
scanf( "%lf", &alpha );
```

Запитання

1. Написати програму, яка вводить з клавіатури три цілих числа та друкує їх середнє арифметичне.
2. Написати програму, яка вводить довжини двох катетів прямокутного трикутника (дійсні числа) та друкує довжину його гіпотенузи.
3. Програма розв'язання квадратного рівняння на с. 27 поки що вміє розв'язувати лише одне рівняння з заздалегідь фіксованими коефіцієнтами. Переробити програму так, щоб користувач вводив довільні коефіцієнти з клавіатури.
4. Написати програму, яка вводить час у звичному форматі
   `години:хвилини:секунди`,
   де години, хвилини та секунди — цілі числа. Кількість годин, хвилин та секунд вводити відповідно у змінні `hour`, `min`, `sec`.
5. Написати програму, яка вводить дату у стандартному форматі
   `число.місяць.рік`.

2.9. Підсумковий огляд

Програма у мові C складається з функцій. В програмі є щонайменше одна функція — головна зі стандартним іменем `main`. Тіло функції береться у фігурні дужки. Всередині тіла функції містяться оголошення змінних та оператори. Оголошення повідомляє транслятору, які імена мають змінні, що потім використовуютсья у тілі функції, та до яких типів вони належать. При оголошенні можна також присвоїти змінній початкове значення.

Імена змінних (а також типів, функціій тощо) є ідентифікаторами. Ідентифікатор — це послідовність літер, цифр, знаків підкреслювання, яка починається не з цифри.

Для того, щоб використовувати в програмі стандартні функції (введення-виведення, тригонометричні тощо) потрібно підключити до програми заголовочний файл, в якому містяться оголошення цих функцій.

Функції при виклику передаються аргументи — дані, які функція буде обробляти. Виклики функцій можуть використовуватися у складі виразів — при обчисленні виразу буде підставлено значення, яке функція повертає.

Серед стандартних функцій є функція виведення, яка дозволяє друкувати текстові повідомлення, а також повідомлення, в які вставлено значення змінних чи виразів. Функція введення дозволяє задати вимоги до формату даних, що вводяться, та повертає кількість значень, які було успішно введено.

Розділ 3

Арифметичні та логічні операції. Структури управління

3.1. Арифметичні операції та перетворення типів

Ключові слова: скорочені операції з присвоюванням, інкремент і декремент, перетворення типу, цілочисельне ділення, операція залишку від ділення, переповнення розрядної сітки.

Описаних вище засобів (арифметичних операцій та оператору присвоювання) в принципі достатньо для того, щоб реалізувати в програмі обчислення за будь-якими формулами. Але запис деяких з них виходить не зовсім зручним. Тому для тих операцій, які надзвичайно часто використовуються у програмах, та які було б незручно кожен раз виписувати через найпростіші операції, в мові C запроваджено скорочені позначення.

Наприклад, дуже часто виникає потреба збільшити або зменшити значення певної змінної на деяке число. За допомогою відомих операцій ця задача вирішується так: `x = x + вираз;`. Скорочена операція збільшення дозволяє записати цю ж дію в більш зручній формі `x += вираз;`. Таким же точно чином працюють операції `-=`, `*=`, `/=` та інші.

Особливо часто в програмі доводиться збільшувати та зменшувати змінну на одиницю. Ця операція, яка стандартними засобами була б записана як `x = x + 1;`, а щойно введеними як `x += 1;`, має ще більш зручні позначення `++` та має назву *інкремент*. Відповідно, існує операція `--`, яка називається *декрементом*. Операції інкременту та декременту можна застосовувати як перед, так і після імені змінної, наприклад `++x` та `++x`. Різниця між цими двома форматами, коли вони застосовуються в складі виразів, роз'яснюється в додатковому розділі А.2. Якщо оператор цілком складається з одного інкременту (декременту), то різниці немає.

Дуже важливими є особливості цілочисельної арифметики. Операція ділення цілого числа на ціле число дає результат цілого типу. При цьому результат ділення не округлюється до найближчого цілого, а просто відкидається дробова частина.

Увага! Обчислення виразу `4/5` дає результат 0, а вираз `4.0/5.0` має значення 0,8. У початківців ця особливість часто стає причиною помилок, які дуже важко знайти в тексті програми.

Якщо є потреба розділити ціле число на ціле число так, щоб отримати точне значення частки (дійсне число), треба перетворити ціле число на рівне за значенням дійсне число. Ця дія називається перетворенням типів. В загальному випадку перетворення типів має вигляд `(тип2) вираз1`, де `вираз1` належить деякому `типу1`, відмінному від `типу2`. Конструкція `(тип2) вираз` є виразом, має `тип2`, а його значенням є значення `виразу1`, приведене до `типу2`.

Отже, для точного ділення цілих чисел слід застосувати перетворення типу **int** до типу **double**, як в прикладі:

```
int x, y;
printf( "введіть␣два␣цілих␣числа␣" );
scanf( "%d␣%d", &x, &y );
printf( "їх␣частка:␣%lf␣\n",
   (double) x / (double) y );
```

Зазначимо, що операція перетворення типу має більш високий пріоритет, ніж операція ділення: спочатку виконуються два перетворення типів, а потім до результатів перетворення застосовується ділення. Отже, хоча операція ділення може виконуватися як над дійсними, так і над цілими числами, але для цих двох типів вона має зовсім різний сенс.

Для цілого типу є також одна особлива операція, яка взагалі не має сенсу для дійсних чисел — операція `%`. Значенням виразу `m%n` є залишок від ділення націло числа `m` на число `n`. Зрозуміло, що залишком може бути число від 0 до `n-1`. Наприклад, вираз `9%4` дає значення 1, оскільки $9 = 2 \times 4 + 1$, вираз `9 % 3` дає значення 0, оскільки $9 = 3 \times 3 + 0$, а вираз `9%10` дає значення 9, бо $9 = 10 \times 0 + 9$ тощо. Звичайно ж, існує і комбінована операція `%=`.

Насамкінець, треба добре розуміти спосіб зображення, зберігання та обробки цілих чисел в машинній пам'яті. Ціле число зберігається у двійковій системі числення, число типу **`int`** на більшості сучасних архітектур займає 4 байти, тобто 32 біт (двійкових розрядів). Старший розряд відрізняється тим, що зберігає знак числа (0 для додатного, 1 для від'ємного), а решта 15 розрядів містять саме число, якщо воно додатнє, або доповнення до абсолютної величини — якщо воно від'ємне.

Це означає, що в тип **`int`** може «вміститися» не як завгодно велике число, а лише числа з певного діапазону. Очевидно, що за допомогою n двійкових розрядів можна закодувати $N = 2^n$ чисел: невід'ємних від 0 до $N - 1$ або від'ємних від -1 до $-N$. Отже, діапазон значень типу **`int`** — від -2147483648 до 2147483647.

Це треба обов'язково мати на увазі, складаючи програму для обчислень з великими за модулем числами. Нехай, наприклад, змінна `x` типу **`int`** має значення 2147483647, тобто найбільше позитивне значення, припустиме для цього типу. Розглянемо операцію інкременту `++x`. З точки зору «чистої» математики слід було б очікувати, що значення змінної `x` стане 2147483648, але через обмеженість розрядної сітки результат буде інший.

Число 2147483647 зображується як $0\underbrace{11\ldots1}_{31}$ (0 в старшому розряді вказує на додатне число). Додавання одиниці призводить до того, що в молодшому розряді утворюється 0 з переносом одиниці в наступний розряд, оскільки в двійковій системі $1 + 1 = 10$. Але у наступному розряді теж стоїть 1, тому перенос одиниці утворює 0 з переносом одиниці у ще більш старший розряд. Такий ланцюжок переносів продовжується 31 раз, поки остання одиниця не перенесеться у найстарший розряд. Отже, після інкременту отримаємо біти $1\underbrace{00\ldots0}_{31}$, що відповідає значенню -2147483648: одиниця в старшому розряді означає знак мінус, а 31 нуль є доповненням до додатного числа 2147483648.

Увага! Якщо при обчисленнях виникне *переповнення* розрядної сітки, тобто виникне число, яке не вміщається в задану кількість біт, то результат роботи програми може виявитися неочікуваним.

Запитання

1. Що означають оператори:
   - `a += 8;`
   - `p *= k + r;`
   - `n--;`
2. Обчисліть значення виразів `14/3`, `14%3`, `15%5`.
3. Нехай змінні `x` та `y` мають тип **`int`** і мають відповідно значення 137 та 100. Як відрізняються значення виразів `x/y` та `(`**`double`**`)x/(`**`double`**`)y`.
4. В розділі було описано, як перетворюється число типу **`int`** до типу **`double`**. Подумайте, як здійснюється протилежне перетворення. Наприклад, дійсне число 36,65 треба перетворити на ціле (втративши дробову частину).
5. Як за допомогою операції `%` перевірити, чи ділиться число `m` на число `n`?
6. Що таке переповнення розрядної сітки і які обмеження це накладає на допустимі в програмі обчислення?

Табл. 3.1. Операції порівняння в мові C

Мова C	Назва	Математика
`==`	Дорівнює	$=$
`!=`	Не дорівнює	$\neq$
`>`	Більше	$>$
`<`	Менше	$<$
`>=`	Більше або дорівнює	$\geqslant$
`<=`	Менше або дорівнює	$\leqslant$

Табл. 3.2. Логічні операції в мові C

C	Назва	Термін	Форма застосування
`&&`	і	Кон'юнкція	`вираз1 && вираз2`
`||`	або	Диз'юнкція	`вираз1 || вираз2`
`!`	ні	Заперечення	`! вираз`

3.2. Логічні операції та операції порівняння

Ключові слова: логічна істина та хиба, операція порівняння, кон'юнкція, диз'юнкція, заперечення.

В програмуванні треба мати можливість не лише проводити обчислення над числовими даними, тобто робити арифметичні операції, але й обробляти *логічні* дані. З логічними даними програма має справу, коли перевіряє, чи виконується деяка умова.

Наприклад, крім звичайного для арифметики питання типу «скільки буде 2×2?», можливо ставити питання типу «чи вірно, що $2 \times 2 = 4$?» або «чи вірно, що $2 \times 2 > 5$?». Іншими словами, можна розглядати $2 \times 2 = 4$ та $2 \times 2 > 5$ як свого роду вирази, і казати про обчислення значень цих виразів. Значенням першого виразу є логічна *істина*, значенням другого — логічна *хиба*.

В мові C логічні значення зображуються за допомогою цілих чисел. А саме, число 0 зображує логічну хибу, а будь-яке відмінне від нуля число зображує логічну істину.

Поки що будемо розглядати найпростіший вигляд логічних виразів, а саме порівняння двох числових виразів:

```
вираз1 операція_порівняння вираз2
```

Операції порівняння показано в таблиці 3.1.

Увага! Значенням операції порівняння є число 1, якщо відповідне співвідношення виконується, та число 0 в іншому випадку.

Увага! Знак `=` означає операцію присвоювання, а операція порівняння на рівність позначається `==`. Часто у початківців плутанина цих двох операцій стає причиною серйозних помилок у програмі. Така помилка тим небезпечніша, що її дуже важко знайти, оскільки помилка полягає в одному-єдиному символі.

Наприклад, якщо змінна `a` має значення 10, то значенням виразу `a >= 0` є число 1 (істина), а значенням виразу `2 * a < a + 5` — число 0 (хиба).

В логіці існують спеціальні операції, або *логічні зв'язки*, які дозволяють конструювати більш складні умови з більш простих, наприклад «число x парне та $x > 20$». Для конкретного значення x значенням цього висловлювання є істина, якщо висловлювання «x парне» істинне і висловлювання «$x > 20$» теж одночасно істинне. Якщо хоча б одне з цих двох висловлювань, або тим більше обидва, хибні, то хибним буде і вказане складне висловлювання.

В мові C є логічні операції, показані в таблиці 3.2. Кон'юнкція та диз'юнкція — бінарні операції, які мають по два операнди, а заперечення — унарна операція, тобто має один операнд. Операції мають таблицю істинності, показану в табл. 3.3.

Додаткова особливість бінарних логічних операцій пов'язана з так званим *лінивим* принципом обчислень: не обчислювати значення операнду, якщо той завідомо не знадобиться. Так, обчислення виразу `вираз1 && вираз2` відбувається таким чином. Спочатку обчислюється

Табл. 3.3. Таблиці істинності логічних операцій

`E1`	`E2`	`E1 && E2`	`E1 || E2`
0	0	0	0
0	не 0	0	1
не 0	0	0	1
не 0	не 0	1	1

`E`	`! E`
0	1
не 0	0

`вираз1`. Якщо його значенням є логічна хиба (число 0), то `вираз2` не обчислюється, а значенням кон'юнкції стає хиба (0). Якщо ж `вираз1` дає логічну істину (будь-яке число, відмінне від 0), то обчислюється `вираз2`, і якщо він дає істину, то значенням кон'юнкції стає число 1, інакше 0.

Обчислення виразу `вираз1 || вираз2` відбувається за схожою схемою. Спочатку обчислюється `вираз1`. Якщо його значенням є логічна істина, то `вираз2` не обчислюється, а значенням диз'юнкції стає 1. Якщо ж `вираз1` дає логічну хибу, то обчислюється `вираз2`, і якщо він дає істину, то значенням диз'юнкції стає число 1, інакше 0.

Запитання

1. Обчислити логічні значення виразів `8 == 8`, `0 == 0`, `1 != 0`, `10 <= 1`, `4 >= -18`.
2. Нехай змінна `x` має значення 5, а змінна `y` — значення 8. Обчислити логічні значення виразів `x == y`, `x != x`, `x <= y`, `x + y == 13` , `x + y != 13`.
3. Обчислити значення виразів
   - `(1 != 0) + (0 == 0)`,
   - `(2 * 2 == 4) - 1`,
   - `y + (x > y)`.
4. Обчислити значення виразів
   - `(x == 0) || (y == 0)`,
   - `(x > 7) && (y > 7)`,
   - `(x > 7) || (y > 7)`,
   - `!( (x > 7) && (y > 7) )`.
5. Що таке ліниве обчислення логічних виразів?
6. Нехай змінні `x` та `y` цілого або дійсного типу. Розглянути вираз `!(x-y)` та переконатися, що він еквівалентний виразу `x==y`, тобто обидва вирази завжди мають однакове значення.
7. Нехай змінні `a` та `b` цілого або дійсного типу. Розглянути вираз `(a>b)*a + (a<=b)*b` та довести, що його значенням завжди є найбільше зі значень `a` та `b`.

3.3. Умовний оператор та розгалужені алгоритми

Ключові слова: **if**, **else**, умовний оператор, вкладені умовні оператори, стиль запису тексту програми та відступи.

На с. 13 було сказано, що для розробки алгоритмів розв'язку задач (якщо задачі не зовсім примітивні) потрібна умовна конструкція, або розгалуження: в деякій точці алгоритму перевіряється певна умова, і в залежності від неї обирається один з двох можливих подальших шляхів. Наприклад, програма розв'язання квадратного рівняння (с. 27) не може коректно обробити випадок, коли дискримінант від'ємний, і намагається в будь-якому випадку продовжити роботу, хоча це завідомо не має сенсу. Більш «розумною» була б така поведінка програми, коли вона перевіряє, чи має місце нерівність $D \geqslant 0$. Якщо нерівність справджується, то

програма обчислює корені, в протилежному випадку програма виводить повідомлення про відсутність дійсних розв'язків. Для цього в мові C існує спеціальний *умовний оператор*, який в загальному випадку має вигляд

```
if( логічний_вираз )
  оператор1;
else
  оператор2;
```

Слова **if** та **else** є ключовими словами мови C, тобто вони не можуть використовуватися для інших цілей, крім умовного оператору (наприклад, не можна називати так змінні або функції).

Виконання умовного оператору відбувається таким чином. Спочатку обчислюється значення виразу в дужках. Якщо це значення є логічною істиною (не дорівнює 0), то виконується перший оператор, в протилежному випадку (логічна хиба, число 0) — виконується другий оператор.

Наприклад, розглянемо задачу: знайти найбільше з двох заданих чисел. Нехай ці числа зберігаються у змінних `a` та `b`, а результат потрібно помістити у змінну `m` та вивести на дисплей. Тоді задачу розв'язує такий оператор

```
if( a > b )
  m = a;
else
  m = b;
printf( "найбільше␣число␣%d\n", m );
```

Треба звернути увагу на те, що виклик функції `printf` в цьому прикладі *не належить* умовному оператору, а стоїть після нього.

Обидві гілки умовного оператору (як слово **if**, так і слово **else**) *зв'язують* лише один наступний оператор. Якщо ж треба включити до умовної конструкції не один, а кілька операторів, то їх треба об'єднати у блок, взявши у фігурні дужки `{...}`.

Наприклад, нехай у програмі потрібно не лише присвоїти значення змінній `m`, але й надрукувати, яке саме з двох чисел виявилося більшим. Тоді програма набуває вигляду

```
if( a > b ) {
 m = a;
 printf( "перше␣число␣більше\n" );
}
else {
 m = b;
 printf( "друге␣число␣більше\n" );
}
printf( "найбільше␣число␣%d\n", m );
```

Нерідко також виникає потреба виконати деякі дії, якщо умова виконується, але не робити нічого (тобто переходити далі до виконання наступного оператора), якщо вона не виконується. Для цього слугує скорочена форма умовного оператору без гілки **else**:

```
if( логічний_вираз )
 оператор;
```

Наприклад, нехай треба взяти значення змінної `x`, знайти його модуль (абсолютне значення) та присвоїти його знову змінній `x`. Згідно означення,

$$|x| = \begin{cases} -x, & \text{якщо } x < 0; \\ x & \text{в іншому випадку.} \end{cases}$$

Тоді задачу розв'язує фрагмент

```
if( x < 0 )
 x = -x;
```

Справді, якщо значення змінної не від'ємне, то воно просто залишиться незмінним, що й треба.

Слід зробити тривіальне зауваження, що в складі гілки умовного оператору може стояти будь-який оператор, в тому числі ще один умовний оператор. Це можна використовувати для побудови складних розгалужених алгоритмів.

Наприклад, в програмі потрібно присвоїти змінній `s` значення 1, якщо змінна `x` має додатнє значення, або значення -1, якщо значення змінної `x` від'ємне, або значення 0, якщо значення змінної `x` є 0.

```
if( x == 0 )
 s = 0;
else {
 if( x > 0 )
  s = 1;
 else
 s = -1;
}
```

Також складні умови можна зводити до більш простих, якщо поєднувати послідовно два умовних оператори. Розглянемо приклад. Нехай потрібно знайти найбільше значення серед трьох змінних, `a`, `b`, `c` та присвоїти результат змінній `m`. Розберемо алгоритм. Для того, щоб знайти найбільше серед трьох чисел, достатньо спочатку знайти відомим способом найбільше з будь-яких двох з них, наприклад, з `a` та `b`, а потім знов знайти найбільше серед двох: знайденого числа та того, що залишилося. Отже, маємо текст

```
if( a > b )
 m = a;
else
 m = b;
/*перший умовний оператор закінчився*/
if( c > m )
 m = c;
printf( "найбільше␣з␣трьох␣чисел␣%d\n", m );
```

Читачу слід звернути увагу на те, що всюди в прикладах оператори, вкладені в умовний оператор, надруковано з відступами. Звичайно ж, транслятору все одно, є відступи в тексті чи немає. Але відступи роблять прозорою логічну структуру програми, тому в програмістському середовищі склалася традиція писати програми саме в такому вигляді. Гарний стиль оформлення тексту програми — неодмінний атрибут програміста-професіонала.

Увага! Неохайно оформлені програмні тексти без належних відступів прийматися не будуть. Студент повинен від самого початку привчитися писати програми гарно.

Запитання

1. Написати програму, яка перевіряє знання таблиці множення: користувач вводить з клавіатури цілі числа, a, b, c. Якщо $a*b = c$, то програма друкує повідомлення «правильно», інакше — «неправильно».

2. Нехай задано кускову функцію

$$f(x) = \begin{cases} x^2, & \text{якщо } x < 2; \\ \frac{x}{2} + 3 & \text{в ішному випадку.} \end{cases}$$

   Написати програму, яка вводить з клавіатури число x та друкує значення $f(x)$.

3. Те ж саме для функції

$$g(x) = \begin{cases} x^2, & \text{якщо } -2 < x < 2; \\ x + 2, & \text{якщо } x \geqslant 2; \\ 2 - x, & \text{якщо } x \leqslant -2. \end{cases}$$

4. Переробити програму розв'язання квадратного рівняння з розділу 2.5 так, щоб вона коректно обробляла випадок від'ємного дискримінанта.

3.4. Оператори циклу

Ключові слова: **while**, **do**, оператор циклу з передумовою та постумовою, тіло циклу, ітерація.

Для втілення ще однієї фундаментальної структури алгоритмів (с. 13) в мові C призначено оператори *циклу*. Оператор циклу з *передумовою* має вигляд

```
while( вираз )
  оператор;
```

Вираз задає умову продовження циклу. Оператор складає *тіло* циклу. Звичайно ж, тіло може бути блоком, тобто послідовністю з кількох операторів, взятих у фігурні дужки. Виконання циклу з передумовою здійснюється за таким правилом:

1. Обчислити значення `виразу`.
   - Якщо воно є логічною істиною, то цикл продовжується: перейти до кроку 2.
   - Якщо воно є логічною хибою, то цикл завершується: перейти до виконання наступного після циклу оператору.
2. Виконати `оператор` (тіло циклу) та перейти до кроку 1.

Отже, як видно з опису, спочатку перевіряється умова, а потім виконується тіло циклу. Один прохід тіла циклу називається *ітерацією*.

Наприклад, програма, наведена нижче, вводить послідовність цілих чисел, поки не зустріне нуль, і підраховує кількість введених чисел, не враховуючи останнього нульового. Розберемо її роботу більш детально.

```
#include <stdio.h>
int main() {
  int x; /* наступне число у послідовності */
  int n; /* кількість введених чисел */
  n = 0;
  printf( "Введіть числа, 0 завершує роботу\n" );
  scanf( "%d", &x ); /* перше число окремо */
  while( x != 0 ) {
    n++; /* наростити лічильник */
    scanf( "%d", &x ); /* ввести наступне число */
  }
  printf( "Було введено %d чисел\n", n );
}
```

Напочатку змінна `n` має значення 0, і це зрозуміло, оскільки досі справді було введено 0 чисел. Перше число з послідовності вводиться до початку циклу. Якщо користувач введе зараз 0, то тіло циклу не виконається жодного разу (див. правило) — і управління буде передано на наступний після циклу оператор, який надрукує, що було введено 0 чисел.

Якщо ж користувач введе не 0, а, скажімо, 23, то виконається тіло циклу. Змінна `n` набуде значення 1 (введено одне число), після цього вводиться друге число. Якщо користувач введе значення 0, то цикл завершиться, а значення змінної `n` залишиться рівним 1. В іншому випадку тіло циклу виконається ще раз (причому значенням змінної `n` стане 2), і так далі.

Другим різновидом є цикл з *постумовою*, в якому спочатку виконується тіло, а потім перевіряється умова продовження. Форма запису та правило виконання такі:

```
do
 оператор;
while( вираз );
```

1. Виконати `оператор` (тіло циклу) та перейти до кроку 2.
2. Обчислити значення `виразу`.
   - Якщо воно є логічною істиною, то цикл продовжується: перейти до кроку 1.
   - Якщо воно є логічною хибою, то цикл завершується: перейти до виконання оператору, що записаний в програмі наступним після циклу.

Для прикладу дамо ще один розв'язок попередньої задачі: підрахувати кількість чисел, які вводить користувач, до першого нуля.

```
1  #include <stdio.h>
2  int main() {
3    int x; /* наступне число у послідовності */
4    int n; /* кількість введених чисел */
5    n = 0;
6    printf( "Введіть␣числа,␣0␣завершує␣роботу\n" );
7    do {
8      ++n; /* наростити лічильник */
9      scanf( "%d", &x ); /* ввести наступне число */
10   } while( x != 0 );
11   n--;
12   printf( "Було␣введено␣%d␣чисел\n", n);
13 }
```

Проаналізувавши роботу програми, читач легко помітить, що при закінченні циклу значенням змінної `n` є кількість всіх введених користувачем чисел, рахуючи і нуль, який завершує послідовність. Оскільки за умовою потрібна лише кількість чисел крім нуля, одразу після циклу значення лічильника треба зменшити на 1.

Нагадаємо, що цикл з передумовою забезпечує виконання тіла 0 або більше разів, а цикл з постумовою виконує тіло 1 або більше разів.

Абсолютно зрозуміло, що в тілі циклу можуть міститися будь-які оператори, в тому числі умовні оператори та інші цикли.

Запитання

- Написати програму перевірки знань таблиці множення. Користувач вводить з клавіатури два цілих числа. Потім робить спробу ввести їх добуток доти, доки не введе його правильно. Програма підраховує кількість невдалих спроб та наприкінці виводить її на екран.
- Написати програму, яка запитує у користувача дійсне число x і друкує значення x^2 та повторює цей процес доти, доки користувач не введе 0.
- Написати програму, яка підраховує суму чисел, що вводить користувач. При введенні нуля або від'ємного числа програма завершує роботу та друкує отримане значення суми.

3.5. Додаткові можливості циклів

Ключові слова: **break**, **continue**, нескінченний цикл, додаткова точка виходу з середини циклу, оператор розриву циклу, цикл з умовою всередині, оператор продовження з наступної ітерації.

Важлива особливість циклів полягає в тому, що при деяких (а можливо і всіх) вхідних даних цикл ніколи не закінчиться. Розглянемо, наприклад, програму

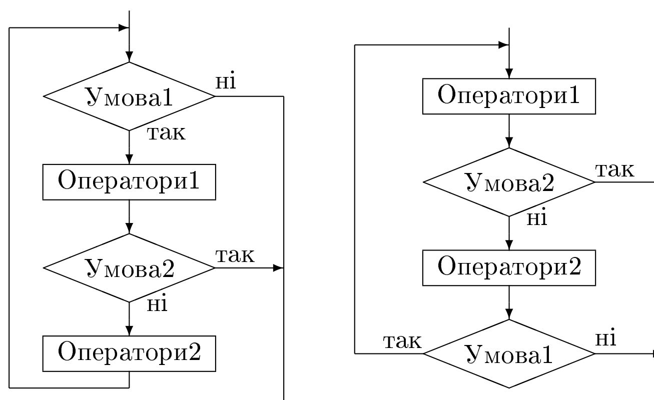

Рис. 3.1. Додаткова точка виходу в тілі циклу

```
int x, s;
s = 0;
while( s < 100 ) {
 printf( "введіть␣число␣" );
 scanf( "%d", &x );
}
```

Цикл має продовжуватися доти, доки значення змінної `s` не перевищить 100. Перед початком циклу ця змінна має значення 0, отже програма увійде до циклу. Але в тілі циклу жоден оператор не змінює значення змінної `s`, отже воно так і залишається рівним 0, тому цикл буде виконуватися вічно.

Увага! Розробляючи програму, програмісту треба особливо ретельно прослідкувати правильність умов завершення циклів.

Іноді зациклювання програми є не помилкою, а саме очікуваною та бажаною її поведінкою. Наприклад, програма автоматичного продажу авіаквитків працює за алгоритмом: отримати замовлення, зменшити на 1 кількість вільних місць на рейс, зняти гроші з банківського рахунку замовника, а потім знов повторити те ж саме. Тобто програма повинна зазначену послідовність дій повторювати весь час в циклі. Такі цикли називають нескінченними.

Для того, щоб зробити нескінченний цикл, достатньо в якості умови поставити логічний вираз, який за будь-яких обставин дає логічне значення істина, наприклад `0 == 0` або `10 > 8`. Якщо згадати, що логічні значення істина та хиба зображуються числом 0 та довільним ненульовим числом, читач легко зрозуміє, що найпростіше умовою циклу зробити просто число 1:

```
while( 1 ) {
  ...
}
```

Оператор циклу забезпечує виконання оператору або послідовності операторів, перевіряючи умову продовження або перед кожною ітерацією (цикл з передумовою), або після неї (з постумовою). Разом з тим, іноді буває зручно завершити цикл достроково, вийти з циклу на середині ітерації. Іншими словами, це означає розмістити в тілі циклу додаткову точку виходу із циклу. Логіку цієї конструкції показано на рис. 3.1.

Для цього слугує спеціальний оператор **break**. Зазвичай оператор **break** в тілі циклу застосовується у складі умовного оператора — тоді умова в операторі **if** грає роль умови виходу з середини циклу (як на наведеній блок-схемі). Якщо під час виконання циклу десь всередині тіла циклу програма виконує оператор **break**, то цикл завершується, і управління передається далі, на наступний оператор.

```
while( умова1 ) {
 оператори1;
```

```
 if( умова2 )
  break;
 оператори2;
}
```

Оператор виходу з середини циклу відкриває нові цікаві можливості. Наприклад, автори мови C безпосередньо заклали в неї цикли, в яких умова продовження перевіряється перед або після кожної ітерації, та не зробили окремого оператору циклу, в якому умова продовження перевіряється всередині ітерації. Попри відсутність в мові C спеціального оператору, таку структуру управління легко змоделювати власноруч як нескінченний цикл з додатковою точкою виходу з середини тіла (треба звернути увагу, що у звичайних операторах циклу умова є умовою продовження, а тут — навпаки, умовою завершення):

```
while( 1 ) {
 оператори1;
 if( умова )
  break;
 оператори2;
}
```

Ще одна можливість циклів — оператор **continue**. Якщо він виконується всередині тіла циклу, то поточна ітерація циклу достроково закінчується, управління передається на точку перевірки умови продовження (перед тілом у циклі з передумовою, після тіла у циклі з постумовою). У програмі

```
while( умова1 ) {
 оператори1;
 if( умова2 )
  continue;
 оператори2;
}
```

під час виконання циклу, якщо виконується `умова2`, то `оператори2` не виконуються, оператор **continue** одразу передає управління на перевірку `умови1`. Для циклу з постумовою — аналогічно.

Слід зазначити, що оператори **break** та **continue** не є необхідними для мови програмування: будь-яку програму, де в циклах використовуються ці два оператори, можна переписати, замінивши їх більш чи менш витонченим використанням операторів **if**. Оператори переривання та продовження циклу запроваджені для зручності, для скороченого позначення тих структур програм, які без них виглядали б більш громіздкими.

Запитання

- Скласти програму, яка обчислює значення функції $y = x^2$ для довільних значень x, які вводить користувач. Програма запитує, чи продовжувати роботу і вводить відповідь у вигляді числа. Якщо користувач відповідає «так» (числом 1), програма запитує та вводить значення x, друкує значення x^2 і повторює процес з початку. Якщо ж користувач відповідає «ні» (числом 0), програма завершує роботу.
- Уявивши, що в мові C немає оператору **break**, змоделювати за допомогою лише операторів **while** та **if** конструкцію

```
while(умова1) {
  оператори1;
  if(умова2) break;
  оператори2;
}
```

- Уявивши, що в мові C немає оператору **continue**, змоделювати за допомогою лише операторів **while** та **if** конструкцію

```
while(умова1) {
  оператори1;
  if(умова2) continue;
  оператори2;
}
```

3.6. Оператор циклу з параметром

Ключові слова: `for`, пробігання по послідовності значень.

Дуже часто в програмах виникає потреба перебрати в циклі всі значення деякої змінної `x`, починаючи від початкового значення `a`, закінчуючи значенням `b`, з кроком `h`, та для кожного значення `x` виконати одну й ту саму послідовність операторів — тіло циклу. Звичайно ж, таку конструкцію логіки побудови програми легко зобразити за допомогою звичайного оператору присвоювання та циклу з передумовою. Але роль та вага даної конструкції настільки велика, що для неї існує спеціальний оператор `for`:

```
for( вираз1; вираз2; вираз3 )
 тіло
```

Тіло може бути задане, звичайно, або одним оператором, або послідовністю операторів, взятою в фігурні дужки. Процедура виконання даного циклу така:

1. обчислити (виконати) `вираз1`;
2. обчислити `вираз2` та діяти в залежності від його значення:
   - якщо значенням є логічна істина (ненульове число), то перейти до кроку 3;
   - якщо значенням є логічна хиба (число 0), то цикл завершується, управління передається на наступний після циклу оператор;
3. виконати `тіло`;
4. обчислити (виконати) `вираз3` та перейти до кроку 2.

Отже, `вираз1` обчислюється лише один раз — перед початком циклу, `вираз2` обчислюється перед початком кожної ітерації, а `вираз3` — наприкінці кожної ітерації.

Як видно з синтаксичної форми оператору та з правила його виконання, він надзвичайно гнучкий, оскільки всі три вирази в заголовку циклу можуть бути в принципі будь-якими. Типове, найбільш традиційне використання оператору `for` полягає в тому, що на місці `виразу1` стоїть присвоювання деякій змінній `x` (параметру циклу) початкового значення, `вираз2` являє собою логічний вираз, порівняння змінної `x` з граничним значенням, а `вираз3` є нарощування змінної `x` на величину кроку. Особливо часто на практиці буває потрібно перебрати значення змінної від 0 до `n-1`, де `n` деяка змінна:

```
for( i = 0; i < n; ++i ) {
  ...
}
```

Треба мати на увазі, що на кожній ітерації нарощування значення (третій вираз) виконується після тіла циклу, а умова продовження циклу (другий вираз) перевіряється перед кожною ітерацією. Припустимо, що змінна `i` вже досягла значення `n-1` після чергової ітерації. Тоді на початку наступної ітерації вираз `i < n` дає значення *істина*, і ще одна ітерація здійснюється. Після неї виконується інкремент, і значення змінної `i` стає рівним `n`. При спробі розпочати ще одну ітерацію вираз `i < n` дасть значення *хиба*, і цикл завершиться.

На відміну від багатьох інших мов програмування, де цикл `for` дозволяє пробігати послідовність значень від якогось члса a до числа b і лише з кроком 1, цикл `for` у мові C набагато гнучкіший і дозволяє найвитонченіші та різноманітні застосування. Розглянемо приклад

```
double x;
for( x = 1000; x > 0.1; x /= 2 ) {
  printf( "%lf\n", x );
}
```

На кожному кроці значення змінної `x` зменшується вдвічі. Отже, змінна пробігає ряд значень 1000, 500, 250, 125, . . .

В тілі циклу **for**, як і в тілі циклів з перед- та постумовами, можуть використовуватися оператори **break** та **continue**.

Увага! Ніколи, крім дуже специфічних випадків, не ставити крапку з комою після заголовку циклу **for** або **while** (з передумовою)! Це улюблена помилка початківців, яка доводить мало не до божевілля: робота програми спотворюється повністю, тоді як текст на перший погляд виглядає правильним. Дуже важко знайти помилку, що складається з одного-єдиного символу, а мова C часто провокує початківця саме до таких помилок. Детальніше про цю помилку див. с. 170.

Запитання

- Уявивши, що в мові C немає оператору **for**, змоделювати його поведінку через оператор циклу з передумовою.
- Скласти програму, яка запитує у користувача кількість n, вводить n дійсних чисел та друкує їх середнє арифметичне.
- Написати програму, яка запитує у користувача ціле число n та підраховує і друкує значення суми $S = 1 + 2 + \ldots + n$.
- Написати програму, яка запитує у користувача два цілих числа, m та n, і друкує на екран m рядків по n зірочок (символів `*`) в кожному.

3.7. Оператор вибору

Ключові слова: **switch**, **case**, розгалуження з багатьма гілками.

Оператор **if** дозволяє розгалужувати програму на дві гілки, одна з яких відповідає істинності, а друга — хибності деякої умови. Часто виникає потреба в розгалуженні алгоритму на n гілок в залежності від значення певного виразу. А саме, якщо вираз e дає значення a_1, виконати оператор P_1, якщо вираз дає a_2, виконати оператор P_2, і так далі.

В принципі це неважко зробити за допомогою оператору **if**, але зазначена логіка алгоритму настільки розповсюджена, що для неї запроваджено спеціальний *оператор вибору*. В загальному випадку він має вигляд

```
switch( вираз ) {
 case константа_1: оператори_1;
 case константа_2: оператори_2;
 ...
 case константа_n: оператори_n;
 default: оператори_0;
}
```

Тут вираз може належати лише якомусь з цілих типів (нагадаємо, що в мові є й інші цілі типи, крім типу **int**). Константи, що стоять після слова **case**, цілочисельні та повинні всі бути різними між собою. Послідовності операторів `оператори_i` не потрібно брати в фігурні дужки. В будь-якій (в тому числі, у всіх або у жодній) з послідовностей `оператори_i` може стояти оператор **break**. Частина **default**: (з наступним за нею оператором) необов'язкова і може бути відсутня. Правила виконання оператору вибору такі:

1. Обчислити значення `виразу`.

2. Порівнювати це значення з кожною по черзі `константою_i` починаючи з першої, доки значення не співпаде. Нехай значення `виразу` співпало з `константою_k`.
3. Виконувати послідовно оператори, починаючи від `операторів_k` і далі, поки не буде досягнуто одне з двох:

   - кінець оператору **switch** (його завершальна фігурна дужка) або
   - оператор **break**.

   У будь-якому з цих двох випадків виконання оператору вибору закінчується, і управління передається на наступний за ним оператор.
4. Якщо значення `виразу` не співпало з жодною `константою_i`, то управління передається на гілку **default**, тобто на `оператори_0`.

Іншими словами, оператор **switch** обчислює значення `виразу` і передає управління на те місце в тілі, мітка-константа якого співпадає зі значенням виразу. Далі оператори виконуються від цього місця до найближчого оператору **break** або до кінця тіла.

Найчастіше на практиці оператор вибору використовують з оператором **break** у *кожній* гілці:

```
switch( вираз ) {
 case константа_1: оператори_1; break;
 case константа_2: оператори_2; break;
 ...
 case константа_n: оператори_n; break;
 default: оператори_0;
}
```

Тоді, згідно наведеного вище правила, процедуру його виконання можна записати так:

0. Обчислити значення `виразу`, нехай це буде α;
1. Якщо $\alpha ==$`константа_1`, виконати `оператор_1`;
2. Якщо $\alpha ==$`константа_2`, виконати `оператор_2`;

....

i. Якщо $\alpha ==$`константа_i`, виконати `оператор_i`;

....

n. Якщо $\alpha ==$`константа_n`, виконати `оператор_n`;

$n+1$. Якщо співпадінь немає, виконати `оператор_0`;

Одним з дуже типових застосувань оператору вибору є організація поведінки програми за принципом меню. Програма в цілому побудована як цикл (будемо називати його головним циклом). На кожній ітерації на екран друкується меню, в якому пункти (команди) мають номери. Користувач вводить номер обраної команди. Далі за допомогою оператору вибору програма аналізує введену команду та виконує відповідний набір дій. Якщо користувач вводить номер, якого в меню немає, то секція **default** друкує повідомлення про неправильну команду.

Варто взяти на озброєння такий прийом. Умовою продовження головного циклу програми є цілочисельна змінна, яку назвемо `flag`. Перед початком головного циклу їй присвоюється значення 1 (логічна істина). Цикл продовжується доти, доки значенням змінної `flag` залишається логічна істина. Тому для організації команди виходу з програми достатньо присвоїти цій змінній значення 0. Заготовку програмного тексту показано в лістингу 3.7.

```
1 #include<stdio.h>
2 int main() {
3   int flag; /* істинний, продовжувати роботу */
4   int cmd; /* введена користувачем команда */
5   /* один раз на початку програми */
6   printf( "Вітаю\n"{} );
7   flag = 1; /* для запуску головного циклу */
8   while( flag ) {
```

```
 9      /* далі друкується меню */
10      printf( "0\tвихід\n" );
11      printf( "1\tновий␣файл\n" );
12      printf( "2\tвідкрти␣файл\n" );
13      printf( "3\tзберегти␣файл\n" );
14      printf( "ВВЕДІТЬ␣КОМАНДУ:␣" );
15      scanf( "%d", &cmd );
16      /* яку команду введено? */
17      switch( cmd ) {
18        case 0: /* команда виходу */
19          flag = 0;
20          break;
21        case 1: /* команда створення нового файлу */
22          /* деякі оператори */
23          break;
24        case 2: /* команда відкриття файлу */
25          /* деякі оператори */
26          break;
27        case 3:/* команда закриття файлу */
28          /* деякі оператори */
29          break;
30          /* введена користувачем команда не
31             співпадає з числами 0, 1, 2 або 3 */
32        default:\\
33                printf( "неправильна␣команда" );
34      } /* закінчився switch */
35    } /* закінчився while */
36    /* один раз при завершенні */
37    printf( "На␣все␣добре\n" );
38  }
```

Запитання

1. Уявивши, що в мові С немає оператору **switch**, змоделювати його поведінку за допомогою операторів **if**.
2. Побудувати програму, яка взаємодіє з користувачем на основі меню та дозволяє для довільного значення x обчислювати значення функції на вибір користувача: $\sin x$, $\cos x$, x^2 та $\sqrt{x}$.

3.8. Підсумковий огляд

Для зручності мова С містить скорочені позначення для часто вживаних операцій вигляду «до значення змінної додати певне значення (або відняти, помножити, розділити) та результат помістити в ту ж змінну». Скорочені позначення є і для операцій збільшення та зменшення змінної на 1.

Цілочисельна арифметика, реалізована на обчислювальній машині, має деякі особливості, які відрізняють її від «чистої» арифметики. Обмеженість розрядної сітки, в якій зберігаються числа, змушує програміста остерігатися переповнення. Ділення цілих чисел виконується націло, з відкиданням дробової частини.

Мова С містить широкий арсенал засобів для управління послідовністю дій, які виконуються в програмі. Ці засоби дозволяють по ходу виконання вирішувати, виконувати чи не виконувати, повторювати багаторазово чи не повторювати ті чи інші оператори.

Для прийняття таких рішень перевіряються умови. Результатом перевірки умови стає логічне значення «істина» або «хиба». В мові С прийнято моделювати їх цілими числами: хибі відповідає число 0, істині — будь-яке інше число.

Найпростіші умови будуються як порівняння значень двох виразів (чи рівні вони між собою, менший чи більший один вираз за інший). Складні умови можна будувати з більш простих за допомогою логічних операцій «і», «або», «ні».

Умовний оператор, що позначається **if**, обчислює значення логічного виразу, якщо воно істинне, виконує один оператор, а якщо хибне — інший.

Оператори циклу з перед- та постумовою повторюють оператор (тіло циклу) доти, доки залишається істинною умова. Цикл з передумовою перевіряє цю умову перед кожною ітерацією, а оператор з постумовою — після кожної ітерації.

Додаткові зручності при програмуванні циклів забезпечують оператор **break**, який негайно перериває цикл, та оператор **continue**, який достроково завершує поточну ітерацію.

Оператор вибору дозволяє розгалужувати алгоритм на довільну кількість гілок. Він обчислює вираз та передає управління на той оператор, константа при якому співпадає зі значенням виразу.

Розділ 4

Розбір прикладів на застосування структурних операторів

4.1. Умовний оператор з простою умовою

Завдання: написати програму, яка вводить з клавіатури координати x та y точки та перевіряє, чи належить ця точка колу (включаючи границю) радіуса 1 з центром в початку координат. Програма повинна надрукувати на екран повідомлення «всередині» або «ззовні» відповідно.

Перш ніж писати програму, потрібно проаналізувати задачу. Слід з'ясувати, за якою формулою можна визначити приналежність точки внутрішній частині кола (включаючи границю). Очевидно, це ті точки, для яких відстань до центру кола менше або дорівнює його радіусу. Оскільки центр кола збігається з початком координат, задача зводиться до знаходження відстані від точки до початку координат. Підставивши відоме значення радіусу, маємо умову

$$\sqrt{x^2+y^2} \leqslant 1.$$

Цю умову можна спростити, позбувшися операції видобування квадратного кореню. Справді, умова істинна тоді і тільки тоді, коли

$$x^2+y^2 \leqslant 1.$$

Математичний зміст задачі проаналізовано. Тепер потрібно переходити до програмної реалізації. Зокрема, треба продумати, які в програмі використовуються змінні та яким типам вони належать. Очевидно, для роботи програми потрібні лише змінні x та y. За умовою задачі їх предметним змістом є координати точки на полощині. Отже, змінні повинні мати тип **double**, оскільки координати можуть бути і нецілими числами.

Оскільки за умовою задачі потрібно вводити значення змінних та друкувати текстові повідомлення, то треба використовувати функції `scanf` та `printf`, а для цього потрібно підключити заголовочний файл `stdio.h`.

Після проведеного розгляду програма набуває такого вигляду:

```
1  #include<stdio.h>
2  int main() {
3    double x, y;
4    printf( "Введіть два координати " );
5    scanf( "%lf %lf", &x, &y );
6    if( x*x + y*y <= 1 )
7      printf( "Всередині\n" );
8    else
9      printf( "Ззовні\n" );
10   return 0;
11 }
```

Запитання

1. Скласти програму, яка перевіряє, лежить точка (x, y) над параболою $y = x^2$ чи під нею. Координати точки вводить користувач.
2. Скласти програму, яка вводить координати точок $A(x_1, y_1)$ та $B(x_2, y_2)$ та перевіряє, чи відстань між ними не перевищує 1.

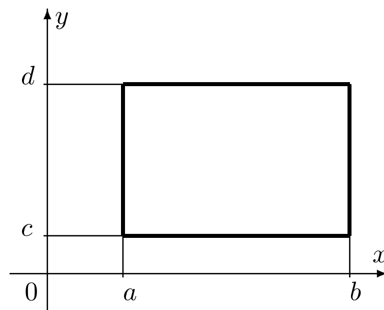

Рис. 4.1.

4.2. Складна умова

Завдання: написати програму, яка розпізнає, чи лежить точка всередині прямокутника (включаючи границю), рис. 4.1.

Легко здогадатися, що точка належить даній області тоді і тільки тоді, коли одночасно виконуються такі чотири умови: $x \geqslant a$, $x \leqslant b$, $y \geqslant c$ та $y \leqslant d$. Згадаємо, що для конструювання складних логічних виразів з більш простих треба використовувати логічні операції. Отже, логічний вираз в умовному операторі буде складатися з чотирьох порівнянь, поєднаних між собою кон'юнкцією. Отримаємо програму:

```
1  #include<stdio.h>
2  int main() {
3    double x, y;
4    printf( "Введіть␣дві␣координати␣" );
5    scanf( "%lf␣%lf", &x, &y );
6    if((x>=a) && (x<=b) && (y>=c) && (y<=d))
7      printf( "Всередині\n" );
8    else
9      printf( "Ззовні\n" );
10   return 0;
11 }
```

Запитання

1. Скласти програму, яка перевіряє, чи лежить точка з координатами (x, y), які вводить користувач, в області між колом рідіуса 1 та колом радіуса 2 з центром в початку координат.
2. Користувач вводить чотири цілих числа a, b, c, d. Скласти програму, яка перевіряє, чи знаходяться ці числа в порядку зростання, тобто чи виконується співвідношення $a \leqslant b \leqslant c \leqslant d$.

4.3. Вкладені умовні оператори

Завдання: написати програму, яка обчислює значення кусково заданої функції $f(x)$. Програма вводить значення x та друкує відповідне значення f.

$$g(x) = \begin{cases} x^2, & \text{якщо } -2 < x < 2; \\ x + 2, & \text{якщо } x \geqslant 2; \\ 2 - x, & \text{якщо } x \leqslant -2. \end{cases}$$

В наведених раніше прикладах програма повинна була розрізнити між двома випадками. В даному прикладі маємо вибір з трьох можливих варіантів. Оскільки умовний оператор мови

С дозволяє розрізняти лише два варіанти, то треба застосувати вкладений умовний оператор. Наприклад, спочатку розрізнити, чи має місце $x \leqslant -2$. Якщо це так, то обчислити відповідне значення (у формулі йому відповідає третій рядок). В іншому випадку треба розпізнати, чи виконується співвідношення $x \geqslant 2$. Якщо так, то маємо випадок, який відповідає другому рядку у формулі. Якщо ж ні, то, оскільки раніше вже з'ясовано, що співвідношення $x \leqslant -2$ невірне, маємо $-2 < x < 2$, тобто першу альтернативу. Нижче наведено текст програми.

```
1  #include<stdio.h>
2  int main() {
3    double x, f;
4    printf( "Введіть␣значення␣аргументу␣" );
5    scanf( "%lf", &x );
6    if( x <= -2 )
7      f = 2 - x;
8    else
9      if( x >= 2 )
10       f = x + 2;
11     else
12       f = x * x;
13   printf( "Значення␣функції␣дорівнює␣%lf\n", f );
14   return 0;
15 }
```

Запитання

Мішень складається з двох кіл з центром в початку координат, перше має радіус 1, друге — радіус 2. Користувач «робить постріл» — вводить координати (x, y) точки A. Скласти програму, яка друкує «відмінно», якщо точка A лежить всередині внутрішнього кола, «добре», якщо між колами, та «мимо», якщо вона лежить ззовні більшого кола.

4.4. Взаємодія кількох умовних операторів

Завдання: написати програму, яка знаходить і друкує на екран максимальне з чотирьох чисел, які вводить користувач.

Позначимо ці числа a, b, c, d. Результат — знайдене максимальне число будемо зберігати у змінній m. Задача допускає кілька різних за логікою розв'язків. Перший варіант полягає в тому, щоб для кожного з чотирьох чисел перевіряти, чи є воно максимальним. Так, число a є максимальним тоді і тільки тоді, коли одночасно виконуються умови $a \geqslant b$, $a \geqslant c$, $a \geqslant c$, число b — тоді, коли істинні умови $b \geqslant a$, $b \geqslant c$, $b \geqslant d$, і т.д. Отже, маємо програму, яка складається з чотирьох умовних операторів, в кожному з яких умову задано кон'юнкцією з трьох порівнянь.

```
1  #include <stdio.h>
2  int main() {
3    double a, b, c, d, m;
4    printf( "Введіть␣чотири␣числа␣" );
5    scanf( "%lf%lf%lf%lf", &a, &b, &c, &d );
6    if((a>=b) && (a>=c) && (a>=d))
7      m = a;
8    if((b>=a) && (b>=c) && (b>=d))
9      m = b;
10   if((c>=a) && (c>=b) && (c>=d))
11     m = c;
12   if((d>=a) && (d>=b) && (d>=c))
13     m = d;
14   printf( "Макс.␣значення␣%lf\n", m );
```

```
15    return 0;
16 }
```

Другий варіант розв'язку. Порівняємо спочатку між собою значення a та b, найбільше з них занесемо у допоміжну змінну p, те ж зробимо зі значеннями c та d, найбільше з них занесемо у змінну q. Тоді залишається порівняти між собою значеня p та q — найбільше з них, очевидно, і буде числом m, яке потрібно знайти. Відповідний програмний текст наведено нижче.

```
1  #include <stdio.h>
2  int main() {
3    double a, b, c, d, m, p, q;
4    printf( "Введіть␣чотири␣числа␣" );
5    scanf( "%lf%lf%lf%lf", &a, &b, &c, &d );
6    if(a>=b)
7      p = a;
8    else
9      p = b;
10   if(c>=d)
11     q = c;
12   else
13     q = d;
14   if(p>=q)
15     m = p;
16   else
17     m = q;
18   printf( "Макс.␣значення␣%lf\n", m );
19   return 0;
20 }
```

Запитання

Будемо називати число x малим за модулем, якщо $|x| < 10$, та великим за модулем, якщо $|x| > 100$. Скласти програму, яка для введеного з клавіатури цілого числа x друкує на екран відповідні ознаки: парне, непарне, велике, мале, від'ємне, додатнє або нуль.

4.5. Цикл з параметром

Завдання: протабулювати функцію. Дано функцію $f(x)$, наприклад $f(x) = x^2 + x + 1$. Програма вводить з клавіатури значення a, b, h, причому $a < b$, $h < b - a$. Програма друкує на екран таблицю значень функції $f(x)$ для всіх значень x, починаючи з $x = a$, з кроком h, поки значення x не досягне числа b. Наприклад, для $a = 3, b = 5, h = 0.2$ програма повинна надрукувати такий текст:

```
f(3.000000) = 13.000000
f(3.200000) = 14.440000
f(3.400000) = 15.960000
f(3.600000) = 17.560000
f(3.800000) = 19.240000
f(4.000000) = 21.000000
f(4.200000) = 22.840000
f(4.400000) = 24.760000
f(4.600000) = 26.760000
f(4.800000) = 28.840000
```

Для перебору значень змінної в заданих границях краще за все скористатися циклом **for**. Нагадаємо, що в цьому операторі потрібно вказати три вирази. Перший виконується один раз перед початком циклу і використовується для присвоювання змінній-параметру початкового

значення, другий вираз обчислюється перед кожною ітерацією та дає логічне значення — «чи продовжувати цикл?», третій вираз виконується після кожної ітерації та використовується для нарощування параметру циклу. Параметром, очевидно, є змінна x, її початковим значенням є a, цикл виконується доти, доки істинне співвідношення $x \leqslant b$, та після кожної ітерації значення x збільшується на величину h.

Друк рядка таблиці не становить труднощів — треба записати у форматному рядку літеру f, відкривальну круглу дужку, специфікатор, замість якого буде підставлено значення x, а далі символи закривальної дужки та рівності, специфікатор, замість якого підставиться значення функції, і, нарешті, перехід на новий рядок. Для того, щоб текст програми виглядав зрозумілішим, доцільно запровадити допоміжну змінну, в яку поміщати на кожній ітерації обчислене значення функції. Результатом наведених міркувань стає текст програми:

```
1  #include <stdio.h>
2  int main() {
3    double a, b, h, x, f;
4    printf( "Введіть␣границі␣та␣крок␣" );
5    scanf( "%lf%lf%lf", &a, %b, &h );
6    for( x = a; x <= b; x+=h ) {
7      f = x*x + x + 1;
8      printf( "f(%lf)␣=␣%lf\n", x, f );
9    }
10   return 0;
11 }
```

Запитання

1. Скласти програму, яка вводить з клавіатури ціле число n та друкує рядок з n зірочок.
2. Скласти програму, яка вводить з клавіатури ціле число n, потім n разів вводить дійсні числа та друкує їх квадрати.

4.6. Цикл з умовним оператором в тілі

Завдання: для заданого цілого числа n знайти всі його дільники. Програма вводить число n з клавіатури і друкує на екран всі його дільники, кожен в окремому рядку.

Дільниками цілого числа n, зрозуміло, можуть бути лише числа з проміжку $1 \leqslant k \leqslant n$. Тому задачу можливо реалізувати так: перебрати всі числа від 1 до n, для кожного з них перевірити, чи є воно дільником n, і, якщо є, надрукувати його.

Перевірку дільності краще за все робити за допомогою операції залишку: число n ділиться на k тоді і тільки тоді, коли залишок від ділення n на k дорівнює 0. Отже, маємо програмний текст:

```
1  #include <stdio.h>
2
3  int main() {
4    int n, k;
5    printf( "Введіть␣ціле␣число␣" );
6    scanf( "%d", &n );
7    printf( "Його␣дільники:\n" );
8    for( k = 1; k <= n; ++k )
9      if( n % k == 0 )
10       printf( "%d", k );
11   return 0;
12 }
```

Табл. 4.1. Таблиця множення

1	2	3	4	5	6	7	8	9
2	4	6	8	10	12	14	16	18
3	6	9	12	15	18	21	24	27
4	8	12	16	20	24	28	32	36
5	10	15	20	25	30	35	40	45
6	12	18	24	30	36	42	48	54
7	14	21	28	35	42	49	56	63
8	16	24	32	40	48	56	64	72
9	18	27	36	45	54	63	72	81

Запитання

Скласти програму, яка вводить цілі числа m та n, потім перебирає по порядку всі числа k у проміжку від m до n і друкує на екран ті з них, для яких $k^2 + 2k + 7$ ділиться на 3.

4.7. Вкладений цикл

Завдання: надрукувати таблицю множення чисел від 1 до 9. Програма повинна надрукувати на екран текст, що складається з 9 рядків та 9 колонок однакової ширини. На перетині i-го рядка з k-ю колонкою повинно стояти число $i \cdot k$. Іншими словами, програма повинна надрукувати табл. 4.1:

На цьому прикладі добре ілюструється ключовий підхід до складання програм, що полягає у розбитті (*декомпозиції*) складної задачі на відносно прості підзадачі (латин. *divide et impera* — розліляй та володарюй).

Уявімо спочатку, що деяке число i вже відоме, і потрібно надрукувати лише один рядок таблиці — рядок чисел однакової ширини, добутків числа i з усіма по черзі числами k від 1 до 9. Очевидно, що для перебору всіх значень k треба застосувати цикл **for**, а для друку чисел з однаковою шириною можна скористатися символом горизонтальної табуляції:

```
for( k = 1; k <= 9; ++k )
    printf( "%d\t", i*k );
printf( "\n" );
```

Справді, семантика оператору **for** забезпечує виконання тіла циклу спершу для значення $k = 1$, потім для $k = 2$ і т.д., а після $k = 9$ цикл завершиться. Функція друку у тілі циклу для кожного k спочатку друкує потрібний добуток $i \cdot k$, а після нього символ табуляції, який пересуває курсор на початок наступної восьмисимвольної колонки екрану. Нарешті, виклик функції друку в третьому рядку відбувається один раз після завершення циклу і переводить курсор на новий рядок.

Отже, маємо фрагмент програмного коду, який друкує один рядок таблиці множення для заданого значення i. Тепер для того, щоб надрукувати всю таблицю, треба застосувати цей фрагмент по черзі для значень $i = 1, 2, \ldots, 9$. Очевидно, перебір значень i робиться за допомогою циклу, а наведений вище фрагмент є тілом цього циклу. Задачу розв'язано, повний текст програми наведено нижче.

```
1 #include <stdio.h>
2 int main() {
3   int i, k;
4   for( i = 1; i <= 9; ++i ) {
5     for( k = 1; k <= 9; ++k )
6       printf( "%d\t", i*k );
7     printf( "\n" );
8   }
```

```
9     return 0;
10 }
```

Даний приклад ілюструє важливу ідіому програмування — *вкладені цикли*, тобто цикли, тіло яких, в свою чергу, є циклом.

Запитання

Скласти програму, яка запитує у користувача числа m та n і друкує прямокутний візерунок розміру $m \times n$, як показано на рис 4.2. Тобто в i-му рядку в j-й позиції має стояти або символ «[», якщо $i + j$ парне число, або символ «]» в іншому випадку.

```
[][][][][]
][][][][][
[][][][][]
][][][][][
[][][][][]
][][][][][
```

Рис. 4.2. Візерунок

4.8. Складні вкладені цикли

Завдання: скласти програму, яка вводить з клавіатури ціле число n та друкує на екран трикутник з зірочок, подібний до зображеного на рис. 4.3 (n рядків в висоту та n символів в ширину).

```
*****
 ****
  ***
   **
    *
```

Рис. 4.3. Трикутник

Знову діємо методом декомпозиції задачі. Надрукувати потрібну за умовою фігуру — значить надрукувати n рядків. Тоді загальна структура основної частини програми повинна мати вигляд

```
/* фрагмент 1 */
for( i = 1; i <= n; ++i ) {
    надрукувати i-й рядок;
}
```

Тепер зосередимо увагу на підзадачі, що стоїть в тілі циклу. Треба надрукувати рядок, який складається з n символів, перші з яких — певна кількість пробілів, а останні — певна кількість зірочок. Друк заданої кількості M однакових символів можна реалізувати циклом, умова якого забезпечує повторення рівно M разів, а в тілі друкується один символ. Після того, як потрібна кількість символів надрукована, потрібно ще перейти на новий рядок. Отже, задача «надрукувати i-й рядок» в першому наближенні розкривається таким чином:

```
/* фрагмент 2 */
for( j = 1; j <= кількість_пробілів; ++j )
    printf( "␣" );
for( j = 1; j <= кількість_зірочок; ++j )
    printf( "*" );
printf( "\n" );
```

Залишається визначити кількість пробілів та зірочок в кожному рядку. Помічаємо, що в першому рядку на рисунку 0 пробілів та 5 зірочок, в другому — 1 пробіл та 4 зірочки, в останньому, п'ятому, відповідно 4 пробіли та 1 зірочка. Легко здогадатися, що в рядку під номером i повинно бути $i-1$ пробіл та $n-i+1$ зірочка. Підставивши ці два вирази у фрагмент 2, а фрагмент 2, у свою чергу, до фрагменту 1, та додавши потрібне оформлення, отримуємо програму

```
1  #include <stdio.h>
2  int main() {
3    int n, i, j;
4    printf( "Введіть␣розмір␣трикутника␣" );
5    scanf( "%d", &n );
6    for( i = 1; i <= n; ++i ) {
7      for( j = 1; j <= i-1; ++j )
8        printf( "␣" );
9      for( j = 1; j <= n-i+1; ++j )
10       printf( "*" );
11     printf( "\n" );
12   }
13   return 0;
14 }
```

Запитання

Скласти програму, яка для введеного користувачем цілого числа n друкує фігуру з n рядків, подібну до зображеної на рис. 4.4:

```
    *
   ***
  *****
 *******
*********
```

Рис. 4.4. Рівнобічний трикутник

4.9. Максимум в послідовності

Завдання: програма вводить послідовність цілих чисел невизначеної заздалегідь довжини. Ознакою кінця послідовності є введення числа 0. Програма друкує на екран найбільше серед введених чисел (рахуючи завершальний нуль — якщо всі введені перед ним числа були від'ємними, то саме цей нуль виявиться найбільшим елементом послідовності).

В реальних застосуваннях дуже часто виникає потреба знайти найбільший чи найменший елемент в деякому наборі, тому ця задача має виняткове велике значення в навчанні програміста-початківця.

Як треба діяти людині, щоб з послідовності чисел, написаних в клітинах на довгій стрічці в довільному порядку, знайти найбільше? Будемо продивлятися стрічку зліва направо, кожен раз дивлячись лише на одне число. На кожному кроці пам'ятатимемо, яке число було найбільшим серед тих, які вже проглянуто. Нехай це буде число m. Тоді на наступному кроці, побачивши нове число x, маємо: після долучення x до розглянутої частини або найбільшим залишиться m, якщо $x < m$, або найбільшим стане x в іншому випадку. В обох випадках знайдене число, найбільше у проглянутій частині стрічки, позначаємо через m і продовжуємо процес.

Справді, якщо через L позначити деяку послідовність чисел, через $L\,x$ — послідовність, утворену з L приписуванням справа ще одного числа x, а через μ позначити найбільший елемент послідовності, то очевидне співвідношення

$$\mu(L\,x) = \max\{\mu(L), x\}.$$

Залишається звернути увагу на початковий крок процесу, коли алгоритм «бачить» лише перше число на стрічці. На цей момент серед усіх досі розглянутих чисел (а їх рівно одне) найбільше знайти дуже легко: це і буде дане число. Отже, маємо алгоритм пошуку найбільшого у послідовності:

1. Покласти $m =$ перше число послідовності.
2. Послідовність закінчилася? Якщо так, то алгоритм завершує роботу, поточне значення m є максимальним елементом послідовності, яке треба було знайти. Якщо ні, продовжувати з наступного кроку.
3. Взяти з послідовності наступний елемент x.
4. Чи $x > m$? Якщо так, то змінити значення m, поклавши $m = x$, якщо ні — не змінювати значення.
5. Перейти до кроку 2.

З наведеного аналізу автоматично випливає програмний текст.

```
1  #include <stdio.h>
2  int main() {
3    int x, m;
4    scanf( "%d", &x );
5    m = x;
6    while( x != 0 ) {
7      scanf( "%d", &x );
8      if( x > m )
9        m = x;
10   }
11   printf( "Максимальний␣%d\n", m );
12   return 0;
13 }
```

Запитання

Застосувавши ту ж загальну схему алгоритму, скласти програму, яка знаходить з точністю до h те значення x, при якому функція $f(x) = \frac{x+8}{x^2+x+1}$ досягає свого найбільшого значення на проміжку $[a, b]$. Числа a, b, h вводить користувач.

4.10. Сума числового ряду

Завдання: скласти програму, яка з заданою точністю ε обчислює наближене значення суми знакозмінного ряду

$$s = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} = \frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \ldots.$$

Користувач вводить з клавіатури величину ε, програма друкує знайдене наближене значення s.

В задачі потрібно знайти суму великої кількості доданків, кожен з яких обраховується за певною формулою. Оскільки у формулі кожен доданок залежить від числа k, яке є номером доданку і пробігає значення $1, 2, 3, \ldots$, то ясно, що обрахунок доданків повинен здійснюватися у циклі.

Далі потрібно відповісти на питання, як, породжуючи в циклі доданки, знайти їх суму. Нехай на кожній ітерації циклу змінна s має зміст «сума тих доданків, що досі були обраховані на всіх попередніх ітераціях», а черговий доданок дорівнює $(-1)^k x$. Тоді при переході

на наступну ітерацію змінна s повинна містити суму всіх доданків, обрахованих і на попередніх, і на цій ітерації. Отже, в тілі циклу змінній s потрібно присвоїти нове значення, яке дорівнює $s + (-1)^k x$.

Тепер, треба продумати, як обчислювати кожен наступний доданок. Зі знаменником труднощів не виникає, але увагу треба звернути на чисельник $(-1)^{k+1}$. Початківці часто роблять велику помилку, коли намагаються втілити в програмі обчислення цього члена безпосередньо, за принципом «як написано у формулі, так і обчислювати в програмі». Насправді цього робити не слід, бо це призведе до значного і, головне, абсолютно зайвого ускладнення програми.

Натомість треба думати не про те, як обчислювати в програмі значення $(-1)^{k+1}$, а відповісти на запитання «яку роль відіграє в математичній формулі цей вираз?». Насправді він слугує для того, щоб знаки доданків чергувалися: якщо поточний доданок додатній, то наступний повинен бути від'ємним. Цього легко добитися, запровадивши в програмі змінну q з сенсом «знак поточного доданку». Цій змінній потрібно присвоїти початкове значення 1 (перший доданок, за формулою, додатній). На кожній ітерації циклу ця змінна домножається на -1 (тобто їй присвоюється нове значення $(-1) \cdot q$. Зрозуміло, що це призводить до чергування знаків $+1, -1, +1, -1, +1, \ldots$.

Нарешті треба з'ясувати, якою буде умова завершення циклу. За умовою задачі треба додати не якусь заздалегідь визначену кількість доданків (тоді задача розв'язувалася б за допомогою циклу **for**), а стільки, скільки треба для досягнення заданої точності. Оскільки ряд знакозмінний і кожен наступний член за модулем менше за попередній, то очевидно, що похибка наближення на кожному кроці циклу не перевищує за модулем поточного доданку. Отже, якщо через x позначено значення доданку на кожній ітерації, то цикл треба повторювати доти, доки $x > \varepsilon$.

В підсумку маємо такий програмний текст:

```
1  #include <stdio.h>
2  int main() {
3    double eps, s=0, x, q=1;
4    int k = 1;
5    printf( "Введіть мале число " );
6    scanf( "%lf", &eps );
7    do {
8      x = 1.0 / k;
9      s += q * x;
10     q *= -1;
11     ++k;
12   } while( x > eps );
13   printf( "Сума %lf\n", s );
14   return 0;
15 }
```

Запитання

1. Написати програму, яка друкує суму дільників введеного користувачем числа n.
2. Написати програму, яка для введеного користувачем цілого числа n друкує значення його факторіалу
$$n! = 1 \cdot 2 \cdot \ldots \cdot n,$$
причому за означенням $0! = 1$. Підказка. Хоча факторіал завжди є цілим числом, для зберігання його значення застосовуйте змінну дійсного типу, оскільки факторіал дуже швидко зростає, і вже для невеликих n значення $n!$ перевищує розрядність цілого типу.
3. Скласти програму, яка вводить з клавіатури дійсне число x та ціле число n і обраховує значення суми[1]

[1] При достатньо великому n значення цієї суми прямує до величини e^x.

$$s = \sum_{k=0}^{n} \frac{x^k}{k!}$$

4.11. Рекурентна послідовність

Числами Фібоначчі називають послідовність чисел F_k, таку, що

1. $F_1 = F_2 = 1$,
2. $F_{k+2} = F_k + F_{k+1}$ для будь-якого $k \geqslant 1$.

Наприклад, $F_3 = F_1 + F_2 = 1 + 1 = 2$, $F_4 = F_2 + F_3 = 3$, і т.д. Отже, маємо нескінченну зростаючу послідовність: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Як видно з цього означення, наступні члени послідовності визначаються на основі вже обчислених попередніх. Такі числові послідовності називають *рекурентними*, задачі такого типу нерідко трапляються в практичному програмуванні, і володіння відповідними алгоритмами необхідне для професіонала.

Завдання: скласти програму, яка вводить з клавіатури число n та друкує n-е число Фібоначчі. Наведемо спочатку текст програми, а потім дамо детальний розбір принципу її роботи.

```
1  #include <stdio.h>
2  int main() {
3    int n, k;
4    double a=1, b=1, c=1;
5    printf( "Введіть␣номер␣" );
6    scanf( "%d", & n );
7    for( k = 1; k <= n-2; ++k ) {
8      c = a + b;
9      a = b;
10     b = c;
11   }
12   printf( "F[%d]␣=␣%lf\n", n, c );
13   return 0;
14 }
```

Змінна n використовується як номер числа Фібоначчі, яке потрібно знайти. Три змінні a, b, c потрібні для обчислення числа Фібоначчі при $n > 2$. Для того, щоб знайти n-й член послідовності, потрібно спершу знайти $(n-1)$-й та $(n-2)$-й члени, а отже и всі числа F_k для $k = 1, \ldots, n-1$. Початківці часто роблять помилку, намагаючись застосовувати масив, в якому зберігаються ці числа. Насправді це зайве ускладнення програми, цілком достатньо трьох змінних.

Справді, будемо будувати послідовність Фібоначчі, починаючи від двох перших членів, поки не досягнемо n-го члену. Два відомі на початку поточної ітерації члени зберігаються у змінних a та b. Зрозуміло, що перед першою ітерацією початкові значення змінних повинні бути $a = 1, b = 1$. На кожному кроці обчислюємо наступний член як суму двох попередніх і поміщаємо отримане значення у змінну c. При переході на наступну ітерацію поточне число займає місце попереднього, а попереднє йде на місце передпопереднього — себто в змінну a присвоюється число, яке раніше було у змінній b, а в змінну b переміщується значення змінної c. Оскільки на першому ж кроці описаного процесу в змінній c отримаємо третій член послідовності, то для того, щоб отримати F_n (при $n > 2$) потрібно зробити $n-2$ ітерації.

Таким чином, ми переконалися, що для $n > 2$ після завершення циклу в змінній n буде міститися число F_n, яке треба було знайти. Залишається розібрати випадок, коли n дорівнює 1 або 2. Звернімо увагу на умову продовження циклу в наведеній програмі. Змінна k слугує лічильником ітерацій, її початкове значення дорівнює 1. Цикл завершує свою роботу при $k > n-2$. Це означає, що якщо введений користувачем номер числа Фібоначчі буде дорівнювати 1 чи 2, то тіло циклу не виконається жодного разу. Щоб і в цьому випадку

потрібне значення F_n опинилося в змінній c, цій змінній присвоюється початкове значення 1, оскільки $1 = F_1 = F_2$.

Хоча числа Фібоначчі є цілими, однак, оскільки послідовність швидко зростає, ці числа в наведеній програмі зберігаються в змінних дійсного типу, щоб уникнути переповнення розрядної сітки.

Запитання

- Скласти програму, яка для введеного користувачем цілого числа n друкує n перших членів рекурентної послідовності, заданої співвідношеннями:

  1. $d_1 = 1$, $d_2 = 3$, $d_3 = 1$.
  2. $d_{k+3} = d_k - \frac{d_{k+1}}{d_{k+2}}$ для $k \geqslant 1$.

- Перший член рекурентної послідовності x_1 вводить користувач, причому $-1,61803 \leqslant x_1 \leqslant 1,61803$. Наступні члени визначаються за формулою $x_{k+1} = 1 - x_k^2$. Програма друкує на екран 20 членів цієї послідовності.

Розділ 5

Функції

Увага! Матеріал, що викладається в цьому розділі, винятково важливий для всіх подальших років навчання. Студентам потрібно вивчити його з особливою ретельністю.

5.1. Поняття функції

Ключові слова: функція, аргумент та результат функції, прототип та тіло функції, виклик функції, повернення значення функцією, **return**.

При програмуванні будь-яких задач, крім найпростіших, постійно виникає потреба виконувати в кількох різних місцях алгоритму одні й ті самі дії над різними значеннями.

Наприклад, нехай потрібно обчислити площу трьох різних трикутників, перший з яких має сторони a_1, b_1, c_1, другий — a_2, b_2, c_2, третій a_3, b_3, c_3. Це можна було б зробити в програмі так[1]:

```
p1 = (a1 + b1 + c1) / 2;
s1 = sqrt(p1 * (p1-a1) * (p1-b1) * (p1-c1));
p2 = (a2 + b2 + c2) / 2;
s2 = sqrt(p2 * (p2-a2) * (p2-b2) * (p2-c2));
p3 = (a3 + b3 + c3) / 2;
s3 = sqrt(p3 * (p3-a3) * (p3-b3) * (p3-c3));
```

Але ж зрозуміло, що кожного разу писати одну й ту саму формулу дуже незручно. Справді, при цьому витрачаються зусилля та час програміста на багатократне повторення, а не на справді творчу роботу. Розбухає текст програми, в ньому стає важко орієнтуватися, а це призводить до ще більших непродуктивних втрат часу. Припустимо, в першій з формул у програмі зроблено помилку (а в процесі написання програми, нажаль, повністю уникнути помилок неможливо) і потім її «клоновано» 10 разів при повторних використаннях. Тоді і виправлення треба внести теж 10 разів. Це тим більше громіздка робота, що треба продивлятися великий текст програми та вручну знаходити там всі випадки застосування цієї формули. Крім того, є небезпека, що в кількох випадках помилку буде виправлено, а в кількох ні, і це особливо небезпечно, бо програміст впевнений, що всі помилки вже виправлено.

Отже, багаторазово описувати в програмі подібні між собою обчислення дуже незручно. Потрібен засіб, який би дозволяв один раз і наперед описати деякий допоміжний алгоритм чи формулу, а потім по мірі потреби звертатися до нього з основного алгоритму. Це дозволило б позбутися всіх щойно зазначених труднощів. Для цього призначений надзвичайно потужний інструмент мови C, а саме функції.

Функцією в мові C є програмна одиниця, яка має *ім'я*, має деяку кількість *аргументів* визначених типів, містить якусь послідовність операторів, що реалізують певний алгоритм, та може дати *результат* певного відомого типу. Функції призначені для того, щоб з інших місць програми (тобто з інших функцій) їх можна було неодноразово *викликати*. Програма у мові C являє собою сукупність функцій, одна з яких функція `main`, з якої починається виконання програми (точка входу в програму).

Для того, щоб мати змогу викликати функцію, транслятор повинен заздалегідь знати її ім'я, тип значення, кількість та типи аргументів. Ці характеристики разом складають *прототип* функції. Прототип нічого не каже про внутрішню будову функції, він описує

[1] Для обчислення площі трикутника за відомими трьома сторонами a, b, c використовується формула Герона $S = \sqrt{p(p-a)(p-b)(p-c)}$, де $p = \frac{a+b+c}{2}$.

лише її зовнішній вигляд (як до неї звертатися). Прототип — це *оголошення* функції, яке повинно бути відоме транслятору перед викликом функції.

Внутрішня будова функції, алгоритм, за яким вона працює та, зокрема, обчислює результат, визначається *тілом* функції. Нарешті, виклик функції складається з її імені та значень всіх аргументів.

В наведеному вище прикладі потрібна функція під назвою `Geron`, до якої можна звернутися, передавши їй три аргументи дійсного типу — довжини трьох сторін трикутника, і отримати від неї у відповідь дійсне число, значення площі цього трикутника. Наведемо приклад програмного тексту:

```
1  #include <stdio.h>
2
3  /* це оголошення функції - прототип */
4  double Geron( double a, double b, double c );
5
6  /* це реалізація функції */
7  double Geron( double a, double b, double c ) {
8    double p; /* локальна змінна */
9    p = (a + b + c) / 2;
10   /* обчислити вираз і повернути результат */
11   return sqrt(p * (p - a) * (p - b) * (p - c));
12 }
13
14 /* головна функція використовує функцію Geron */
15 int main() {
16   double u, v, w;
17   double s;
18   printf( "введіть␣сторони␣трикутника␣" );
19   scanf( "%lf␣%lf␣%lf", &u, &v, &w );
20   /* викликається функція Geron.
21      До аргументів a, b, c передаються значення
22      змінних u, v, w. Результат виклику присвоюється
23      змінній s */
24   s = Geron( u, v, w );
25   printf( "площа␣трикутника␣%lf\n", s );
26   /* викликається функція Geron.
27      До аргументів a, b, c
28      передаються константи */
29   s = Geron( 10.3, 8.1, 9.7 );
30   printf( "площа␣2-го␣трикутника␣%lf\n", s );
31   /* викликається функція Geron.
32      До аргументів передаються
33      значення виразів */
34   s = Geron( u + 10.3, v + w, w * 1.7 );
35   printf( "площа␣3-го␣трикутника␣%lf\n", s );
36   return 0;
37 }
```

Цей приклад ілюструє такі основні правила (в наступному підрозділі будуть дані більш детальні теоретичні пояснення). Прототип функції закінчується крапкою з комою. Реалізація функції складається з заголовку, який повторює прототип, та тіла у фігурних дужках. В тілі функції можуть оголошуватися *локальні* змінні — такі змінні доступні лише для цієї функції та існують лише під час її виконання. По оператору **return** робота функції завершується, а результатом виклику функції стає значення відповідного виразу. Три виклики функції `Geron` в головній функції ілюструють, що аргументи, що передаються при виклику, можуть бути значеннями змінних, константами, або взагалі значеннями довільних виразів. Значення, яке функція виробила за допомогою оператора **return**, може присвоюватися змінним.

Запитання

1. Для чого потрібні функції в програмах?
2. Для чого слугує прототип функції, яку він несе інформацію для транслятора?
3. Що таке аргументи функції?
4. Як розуміти фразу, яка закріпилася в програмістській термінології, що функція *повертає значення*?
5. Що таке тіло функції, що в ньому міститься?
6. Запишіть прототипи будь-яких функцій, що мають один аргумент цілого типу, два аргументи дійсного типу, два аргументи, один з яких цілого типу, а другий дійсного.
7. Для чого призначений оператор **return**?
8. Спробуйте оголосити, реалізувати функцію, яка приймає два аргументи цілого типу та повертає ціле число: найбільше значення з двох аргументів. Напишіть програму, в якій ця функція викликається.
9. Напишіть функцію, яка має один аргумент `n` цілого типу, запитує користувача та вводить з клавіатури `n` дійсних чисел та повертає їх суму. Скласти функцію `main`, з якої ця функція викликається.

5.2. Важливі додаткові відомості

Ключові слова: прототип функції, порожній тип **void**, повернення спеціального значення як спосіб повідомити про помилку.

Прототип починається з типу значення функції, потім йде ім'я функції, а далі в дужках перераховуються типи та (необов'язково) імена аргументів, розділені комою:

```
тип_знач ім'я_функ(тип_арг_1,...,тип_арг_n);
```

Якщо програма невелика за обсягом та весь її текст розміщується в одному файлі, то зручніше за все всі прототипи зібрати на початку, одразу після директив **#include** та перед усіма реалізаціями функцій. Якщо програма складається з кількох *модулів*, прототипи треба розмістити у власноруч створених заголовочних файлах (такі файли мають розширення `.h`) та підключати їх директивою **#include** — див. розділ 12.

Нерідко виклик функції здійснюють не для того, щоб отримати від неї деяке обчислене значення, а для того, щоб виконати дії над пристроями. Така функція за своїм призначенням просто нічого не повинна повертати. Наприклад, функція, яка приймає один аргумент, номер пункту меню, та друкує на екран текст розширеної підказки по даному пункту. Виникає питання, як позначити в прототипі функції відсутність значення, що повертається.

Нерідко також функція за своїм призначенням не повинна мати жодного аргументу (така функція, може брати потрібні їй дані з файлу, з клавіатури). Наприклад, функція, яка друкує на екран меню разом з запрошенням ввести команду, та повертає введений користувачем номер команди. Тоді виникає питання, як у прототипі позначити відсутність аргументів.

Для цих цілей в мову C запроваджено спеціальний *порожній* тип **void**. В розділі 1.2 було сказано, що тип — це множина допустимих значень (об'єктів даних) в поєднанні з сукупністю операцій, які над цими значеннями можна виконувати (с. 11). Тип **void** виділяється тим, що в ньому немає жодного допустимого значення, та немає жодної операції. Іншими словами, це вироджений випадок типу даних. Зрозуміло, що **void** не можна назвати справжнім типом, скоріше це умовний спосіб позначити відсутність будь-якого типу.

Щоб позначити, що функція не повертає значення, треба написати, що вона повертає значення типу **void**. Точно таким же чином, щоб позначити, що функція не має аргументів, треба вказати у прототипі, ніби вона має один аргумент типу **void**[1]; також функцію без аргументів можна позначити парою дужок, між якими немає аргументів:

[1] У розділі 7 буде описано ще одне важливе застосування типу **void**

```
void ff( int, int ); /* не повертає значення */
int gg( void ); /* без аргументів */
void hh(); /* не повертає, без арг. */
```

Якщо функція повертає значення, то в її тілі, звичайно, повинен бути реалізований алгоритм, який обчислює це значення, та оператор, що має форму **return** `вираз;`, а якщо функція не повертає значення (має тип результату **void**), то оператор повернення скорочується до **return**;.

Коли виконання доходить до оператору **return** `вираз;`, він обчислює значення `виразу` і одразу припиняє виконання функції. Значення `виразу` передається в те місце програми, з якого функцію було викликано.

Так, наведена нижче функція приймає два аргументи, цілі числа, знаходить найбільше з цих двох чисел та повертає знайдене значення:

```
1 int maxof2( int x, int y ) {
2   int m;
3   if( x > y )
4     m = x;
5   else
6     m = y;
7   return m;
8 }
```

Оператор **return** може стояти в будь-якому місці функції. В більшості випадків його розміщують наприкінці тіла функції, але нерідко, якщо алгоритм функції доволі складний і має логічну структуру «якщо виконується умова A_1, повернути значення e_1, а якщо виконується умова A_2, повернути значення e_2», то застосовується кілька операторів повернення в складі умовних операторів. Наприклад, наведену вище функцію можна переписати у такому вигляді, зекономивши три рядки коду:

```
int maxof2( int x, int y ) {
  if( x > y )
    return x;
  return y;
}
```

Зверніть увагу, що в умовному операторі відсутня гілка **else**: в ній немає потреби, бо оскільки оператор **return** негайно завершує функцію, то якщо виконання функції дійшло до оператору **return** `y`, то це само по собі означає, що умова `x > y` виявилася хибною.

Наступний приклад ілюструє, як додатковий оператор **return** дозволяє захистити функцію від передачі недопустимого аргументу. Розглянемо текст функції обчислення факторіалу:

```
1 double factorial( int n ) {
2   int i, p;
3   if( n < 0 )
4     return -1;
5   p = 1;
6   for( i = 1; i <= n; ++i )
7     p *= i;
8   return p;
9 }
```

Відомо, що поняття факторіалу визначене для невід'ємних чисел, вираз $n!$ не має сенсу при $n < 0$. Уявімо, що операторів **if**-**return** напочатку функції немає, тоді помилковий виклик функції `factorial` з передачею від'ємного значення аргументу міг би призвести до неприємних наслідків. Тому «запобіжний клапан» на вході функції в подібних випадках виявляється корисним.

Студентам також слід звернути увагу на застосований тут прийом з поверненням числа −1. Можливі значення факторіалу при допустимих значеннях аргументу, звичайно ж, додатні. Повернення завідомо недопустимого значення тут грає роль сигналу про помилку — функція у такий спосіб повідомляє, що не може обчислити потрібне значення з вини того, хто викликав її з неправильним аргументом. Викликавши цю функцію, треба перевірити повернутий результат, чи не дорівнює він −1, і якщо не дорівнює, значить ним можна користуватися у подальших обчисленнях. Цей прийом з використанням особливого значення в ролі повідомлення про помилку доволі традиційний для мови C.

Запитання

1. Що таке тип **void**, як він використовується, чим відрізняється від усіх інших типів?
2. Поміркуйте, чи можна оголошувати в програмі змінні типу **void**. Відповідь обгрунтувати (в тексті розділу прямої відповіді в явному вигляді немає, але є всі необхідні дані, на основі яких вдумливий читач може дати відповідь самостійно).
3. Написати функцію, яка приймає три аргументи, сторони трикутника, та повертає його площу. Передбачити випадок, коли трикутник з тикими сторонами не існує, в цьому випадку повернути спеціальне значення.
4. Переробити приклад програми на с. 46 так, щоб в окремі функції були винесені: друк меню, запрошення та введення команди користувача, обробки кожної команди меню, крім виходу (функції-обробники команд меню нехай поки що просто друкують повідомлення типу "обробляється команда відкриття файлу". Це так звані заглушки, які застосовуються замість справжніх робочих функцій, поки програма в стадії розробки).

5.3. Локальні та глобальні змінні

Ключові слова: локальні змінні, глобальні змінні, область видимості змінної, час життя змінної.

Змінні, оголошені в тілі функції, називаються локальними. Поведінка та всі основні властивості аргументів функції такі ж, як і у локальних змінних: все, що в цьому розділі сказано про локальні змінні, справедливо і для аргументів. Локальні змінні цілком належать тій функції, в якій оголошені. Інші функції цих змінних «не бачать», тобто не можуть жодним чином до них звернутися — ні взяти, ні присвоїти значення. Тому кажуть, що *областю видимості* локальної змінної є та функція, в якій вона оголошена.

Наприклад, в наведеному тексті є помилка:

```
void ff() {
  int x; /* видима лише у функції ff */
...
}

void gg() {
  x = 0; /* у функції gg змінної x немає! */
}
```

Крім того, *час життя* локальної змінної є час, поки виконується функція, в якій вона оголошена. В той момент, коли програма входить у функцію, її локальні змінні створюються, тобто під них виділяється пам'ять. Локальні змінні існують у пам'яті, поки продовжується виконання тіла функції, а при виході з неї знищуються. Звернімося до наведеного вище прикладу: змінної під іменем `x` просто не існує в пам'яті машини до того моменту, поки не буде викликана функція `ff`.

Змінні, оголошені в двох різних функціях, навіть коли мають однакові імена, жодним чином не взаємодіють, не перекриваються між собою. Наприклад, у фрагменті програми

```
void ff() {
  int x;
  ...
}

void gg() {
  int x;
  ...
}
```

змінна `x` з функції `ff` та змінна `x` з функції `gg` — це дві зовсім різні змінні, між якими немає нічого спільного.

Увага! Чомусь у студентів іноді виникають труднощі з розумінням цього цілком очевидного правила. Розглянемо аналогію: оцінка Петренка та оцінка Іваненка з програмування — дві різні величини. Хоча обидві мають ім'я «оцінка з програмування», але вони локальні для двох різних людей, область видимості кожної — залікова книжка одного або іншого студента. Присвоювання будь-якого значення змінній «оцінка Іваненка з програмування» не змінює оцінку Петренка.

Звідси випливає важливий наслідок. Нехай над великою та складною програмою працює бригада програмістів, і кожен з них пише свою функцію. Тоді, роблячи у своїй функції локальну змінну та обираючи для неї ім'я, можна не турбуватися про те, чи не назвав хтось іще свою змінну таким же іменем у своїй функції. Завдяки локальній області видимості функції стають *незалежними* одна від одної.

Змінні в мові C можуть оголошуватися не лише всередині функції, а ще й за межами тіл функції (зазвичай перед усіма тілами функцій). Такі змінні називаються *глобальними*, на відміну від локальних, їх час життя — весь час виконання програми, а область видимості — тіла всіх функцій у тому ж модулі[1], що стоять після оголошення даної змінної.

Іншими словами, глобальну змінну «бачать» всі функції, і всі можуть їй присвоювати значення. Наприклад, у програмі

```
1  int x; /* глобальна змінна */
2  void f();
3  void g();
4
5  void main() {
6    x = 0;
7    g();
8    f();
9    g();
10 }
11
12 void f() {
13   x = 8;
14 }
15
16 void g() {
17   printf( "%d\n", x );
18 }
```

три функції (`main`, `f`, `g`) мають спільний доступ до змінної `x`. Функція `main` спочатку присвоює змінній `x` значення 0, потім викликає функцію `g`, яка друкує значення змінної, тобто 0. Далі функція `main` викликає функцію `f`, яка тій же змінній присвоює значення 8. Нарешті, Функція `g` знов друкує значення змінної `x`, яке тепер дорівнює 8.

Увага! Виклик функції завжди позначається круглими дужками, в яких стоять вирази, значення яких передаються (підставляються) замість аргументів. У частковому випадку,

[1] див. розділ 12, поки що ми маємо справу з програмами, що складаються лише з одного модуля

коли функція не має аргументів, круглі дужки все одно потрібні, тоді виклик має вигляд `g()`.

Увага! Порядок запису тіл функцій в програмі не має жодного значення. Функції виконуються не в тому порядку, як записані їх тіла, а в тому, в якому вони викликаються, прямо чи непрямо, з функції `main`. Якщо це не зрозуміло, необхідно продовжувати перечитувати посібник з початку і до моменту розуміння.

Увага! Хороший стиль програмування, відпрацьований багаторічним досвідом програмістів-професіоналів, вимагає *ніколи не використовувати глобальні змінні*, якщо немає на те *виняткової* потреби.

Справа в тому, що спільний доступ кількох функцій до глобальної змінної призводить до надто сильного зв'язку між функціями, надмірно тісної взаємодії. Як наслідок, навіть мала зміна в одній з таких функцій може відбитися на всіх інших, втрачається можливість різним програмістам працювати над різними функціями окремо. Натомість, взаємодію між функціями слід здійснювати через передачу аргументів та повернення значення.

Запитання

1. Що таке локальна змінна? Де вона оголошується?
2. Що таке глобальна змінна? Де вона оголошується?
3. Що таке область видимості змінної?
4. Яка область видимості локальної та глобальної змінної?
5. Що таке час життя змінної?
6. Який час життя локальної та глобальної змінної?
7. Чи може одна функція присвоїти значення локальній змінній іншої функції?
8. Чому небажано використовувати глобальні змінні?
9. У функції `f` оголошено змінну **`int`** `x`, і у функції `g` оголошено змінну **`int`** `x`. Це одна й та сама змінна чи дві різних? Чи є між цими змінними взаємодія?

5.4. Виклик та передача значень аргументів

Ключові слова: виклик функції, локальна змінна, створення та знищення локальних змінних.

Розглянемо детальніше, як відбувається виклик функції. Нехай функція `f`, має аргументи `x_1`, ..., `x_k` і локальні змінні `a_1`, ..., `a_l`. Функція `g` має аргументи `y_1`, ..., `y_m`, а також локальні змінні `b_1`, ..., `b_n`. Припустимо, для простоти, що типи всіх змінних, аргументів та значень цілі.

Нехай в тілі функції `f`, є виклик функції `g` з виразами `e_1`, ..., `e_m` на місці аргументів. Такий виклик *сам є виразом*, зокрема може бути правою частиною оператору присвоювання або входити до складу інших виразів.

Нарешті, нехай у тілі функції `g` є оператор **`return`** `e`, де `e` — вираз.

```
int f( int x_1 , ..., int x_k );
int g( int y_1 , ..., int y_m );

int f( int x_1 , ..., int x_k ) {
  int a_1 , ..., a_l;
  ...
  ...g(e_1 , ..., e_m)...
  ...
}

int g( int y_1 , ..., int y_m ) {
```

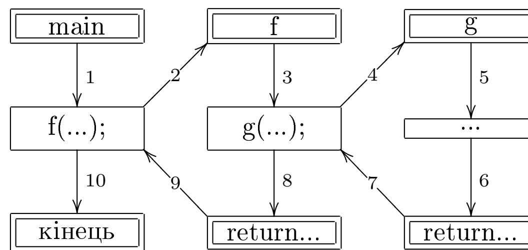

Рис. 5.1. Процес виконання програми з викликами функцій

```
  int b_1, ..., b_n;
  ...
  return e;
}
```

Нехай виконується тіло функції `f`, і управління дійшло до виразу `g(e_1,...,e_m)`. Тоді послідовність дій обчислювальної машини така:

1. Значення виразів `e_i` обчислюються та запам'ятовуються;
2. Локальні змінні `a_j` та аргументи `x_j` функції `f` перестають бути видимими. Вони продовжують існувати в пам'яті машини, але, так би мовити, в «замороженому» стані;
3. Створюються локальні змінні `b_j` та аргументи `y_j` функції `g` — для них виділяється пам'ять;
4. Значення виразів, обчислені на кроці 1, поміщаються у змінні `y_j`;
5. Розпочинається виконання операторів, що складають тіло функції `g`;
6. Виконання доходить до оператору **return** `e`, значення виразу `e` обчислюється та запам'ятовується;
7. Знищуються локальні змінні `b_j` та аргументи `y_j` — пам'ять, яку вони займали, звільняється;
8. Локальні змінні `a_j` та аргументи `x_j` функції `f`, що були «заморожені» на кроці 2, відновлюються, «розморожуються»;
9. Управління повертається до функції `f`, до того самого місця, звідки викликано функцію `g`. При цьому значенням виразу `g(e_1,...,e_m)` стає значення, яке було обчислено на кроці 6;
10. Продовжується виконання тіла функції `f`.

На рис. 5.1 схематично показано послідовність кроків, якщо з функції `main` викликається функція `f`, а з неї, в свою чергу, функція `g`.

Запитання

Нехай дано програму

```
1  int maxof2( int x, int y );
2  int main() {
3    int a, b;
4    a = 7;
5    b = 12;
6    c = maxof2( a, b );
7    printf( "%d\n", c );
8    return 0;
9  }
10 int maxof2( int x, int y ) {
11   int m;
```

```
12   if( x > y )
13     m = x;
14   else
15     m = y;
16   return m;
17 }
```

1. Застосовуючи наведені вище правила, детально опишіть процес виконання програми.
2. Що буде надруковано на екрані в результаті її виконання?
3. Чи змінилася б поведінка програми, якщо б локальна змінна у функції `maxof2` називалася `a`, а не `m`?

5.5. Розбір прикладу виклику функції

Розглянемо тепер процес виклику функцій на конкретному прикладі. Наведена нижче програма вводить з клавіатури два цілих числа, m та n, і друкує на екран значення

$$\frac{(m+1+n+1)!}{(m+1)\cdot(n+1)}.$$

Введення вихідних значень та друк результату здійснюється у функції `main`, обчислення значення дробу

$$\frac{(x+y)!}{x\cdot y}$$

(при $x = m+1$, $y = n+1$) винесено у функцію `fract`, яка, в свою чергу, викликає функцію `factorial` для обчислення факторіалу.

```
1  double fract( int, int );
2  double factorial( int );
3
4  int main() {
5    int m, n, s;
6    scanf( "%d %d", &m, &n );
7    printf( "%lf\n", fract(m+1, n+1) );
8    return 0;
9  }
10
11 double fract( int x, int y ) {
12   double t;
13   t = factorial(x + y) / (x * y);
14   return t;
15 }
16
17 double factorial( int n ) {
18   int i;
19   double p;
20   p = 1;
21   for(i = 1; i <= n; ++i )
22     p *= i;
23   return p;
24 }
```

Опишемо детально кожен крок виконання програми, припустивши, що користувач вводить числа 1 та 3. На момент безпосередньо перед викликом функції `fract` в рядку з номером 7 машина має стан пам'яті

Змінні функції `main` | m : 1 | n : 3 |

Далі обчислюються значення виразів `m+1` та `n+1`, які стоять на місці аргументів, відповідно, числа 2 та 4. Локальні змінні `m` та `n` функції `main` приховуються, а натомість створюються змінні `x`, `y`, `t`. Обчислені перед цим значення 2 та 4 одразу присвоюються змінним `x`, `y`, а значення змінної `t` поки що невизначене (в ній може знаходитися будь-яке непередбачуване число, яке випадково опинилося в пам'яті на тому місці, куди потрапила змінна `t`). Отже, маємо стан пам'яті:

Змінні функції `fract`	`x : 2`	`y : 4`	`t : –`
Заморожені з функції `main`	`m : 1`	`n : 3`	

Виконання тіла функції `fract` одразу ж призводить до виклику функції `factorial`. Діючи за наведеними правилами, маємо, що спочатку обчислюється та запам'ятовується значення виразу `x+y`, воно дорівнює 6. Змінні `x`, `y`, `t` функції `fract` приховуються, а створюються змінні `n`, `i`, `p` функції `factorial`. Значенням змінної `n` одразу ж стає обчислене число 6, значення двох останніх змінних поки що невизначені. Стан пам'яті утворюється такий:

Змінні функції `factorial`	`n : 6`	`i : –`	`p : –`
Заморожені з функції `fract`	`x : 2`	`y : 4`	`t : –`
Заморожені з функції `main`	`m : 1`	`n : 3`	

Увага! Ще раз нагадаємо, що 1) жодна функція не може хоч би як «зіпсувати» локальні змінні іншої функції і 2) коли з однієї функції викликається інша функція, локальні змінні першої функції стають недоступними. В даному прикладі змінна (точніше, аргумент) `n` з функції `main` та змінна `n` з функції `factorial` — *це дві зовсім різні змінні!*

Виконання тіла функції `factorial` за кілька кроків призведе до показаного нижче стану пам'яті (студенту пропонується переконатися в цьому самостійно). Зрозуміло, що зміни могли відбутися лише у верхньому рядку, в локальних змінних тієї функції, яка в даний момент виконується.

Змінні функції `factorial`	`n : 6`	`i : 7`	`p : 720`
Заморожені з функції `fract`	`x : 2`	`y : 4`	`t : –`
Заморожені з функції `main`	`m : 1`	`n : 3`	

У цьому стані виконується оператор **`return`** `p`. Обчислити значення виразу `p` означає просто взяти значення змінної, число 720. Локальні змінні та аргумент функції `factorial` знищуються, а змінні функції `fract` відновлюються. Продовжується перерване, або призупинене виконання тіла функції `fract`.

Виклик функції `factorial` стояв у функції `fract` у складі виразу

```
factorial(x + y) / (x * y)
```

Значенням підвиразу `factorial(x + y)` стає число 720, отже значенням усього виразу є число $\frac{720}{2 \cdot 4} = 90$, яке й присвоюється змінній `t`. В результаті виходить стан пам'яті

Змінні функції `fract`	`x : 2`	`y : 4`	`t : 90`
Заморожені з функції `main`	`m : 1`	`n : 3`	

У цьому стані виконується оператор **`return`** `t`, отже функція `fract` повертає значення 90, яке (після знищення локальних змінних функції `fract`, відновлення змінних та продовження виконання функції `main`) передається як аргумент до функції `printf` та друкується на екран.

Запитання

Скласти програму, яка запитує у користувача число n та

- якщо $n < 1$, завершує роботу,
- в іншому випадку друкує на екран n-е просте число та повторює все знов.

Пошук n-го простого числа реалізувати у вигляді функції. Алгоритм цієї функції перебирає цілі числа і для кожного перевіряє, чи просте воно, та підраховує всі прості, які трапилися при переборі. Розпізнавання того, чи є число x простим, реалізувати у вигляді допоміжної функції, яка повертає логічну хибу або логічну істину.

5.6. Рекурсія

Ключові слова: пряма та непряма рекурсія, рекурсивна функція, рекурентні означення.

Як було сказано вище, логічна структура програми в мові C зводиться до того, як функції викликають одна одну. Кожна функція відповідає певній підзадачі, тому виклик функції `g` з тіла функції `f` означає, що для вирішення задачі f потрібно вирішити допоміжну підзадачу g.

Рекурсивною називають таку функцію, яка викликає сама себе. Це означає, що для вирішення деякої задачі потрібно серед допоміжних підзадач вирішити таку саму задачу, тільки з іншими значеннями параметрів. Розрізняють пряму та непряму рекурсію. Пряма рекурсія полягає в тому, що функція `f` безпосередньо викликає сама себе, а непряма — в тому, що функція `f_1` викликає деяку функцію `f_2`, та в свою чергу викликає функцію `f_3` і так далі, нарешті функція `f_n` викликає функцію `f_1`.

Рекурсивні функції добре підходять для реалізації методу математичної індукції та для програмного моделювання математичних понять, які мають *рекурентні означення*. Спрощено, рекурентним означенням деякого поняття A називають таке означення, в якому використовується саме це поняття A.

Це зручно розглянути на прикладі факторіалу. Факторіалу можна дати таке означення:

$$n! = \begin{cases} 1 & \text{якщо } n = 0, \\ (n-1)! \cdot n & \text{для } n > 0 \end{cases}$$

Це означення розпадається на два правила: перше описує *тривіальний* випадок, те значення аргументу, для якого значення факторіалу можна отримати найпростіше, одразу, не роблячи жодних обчислень. Друге правило каже, як звести складну задачу знаходження факторіалу числа n до простішої підзадачі: знаходження факторіалу меншого числа $n-1$.

Звернімо увагу: в означенні факторіалу використовується (у другому правилі) саме ж поняття факторіалу. Для вирішення задачі про знаходження факторіалу залучається допоміжна підзадача знаходження факторіалу, хоча й — це також дуже важливо — для меншого значення аргументу (тобто підзадача виявляється простішою за основну задачу). Отже, наведене означення факторіалу — рекурентне.

Випишемо в деталях процес застосування цього рекурентного означення. Знайдемо, наприклад, значення 4!. Підставивши 4 замість n, помічаємо, що перше правило з означення застосувати неможна, бо $4 \neq 0$. Тому треба застосувати друге правило:

$$4! = (4-1)! \cdot 4 = 3! \cdot 4 =^{\star} .$$

Чи знаємо ми значення 3! в цьому виразі? Ні, тому обчислення даного виразу відкладаємо на потім (це позначено зірочкою) і займемося обчисленням допоміжного значення 3! Покладемо для цього $n = 3$ і знайдемо факторіал за другим правилом:

$$3! = (3-1)! \cdot 3 = 2! \cdot 3 =^{\star\star} .$$

При обчисленні факторіалу числа 3 за наведеним правилом ми знов стикнулися з потребою обчислити в допоміжних цілях факторіал іншого числа, а саме 2. Оскільки це значення нам поки невідоме, відкладемо обчислення 3! (позначено двома зірочками) та будемо обчислювати факторіал числа $n = 2$. Знов застосовується друге правило.

$$2! = (2-1)! \cdot 2 = 1! \cdot 2 =^{\star\star\star} .$$

Точно таким же чином, обчислення 2! звелося до обчислення допоміжного значення 1!. Тому потрібно перервати процес обчислення 2! та зайнятися обчисленням

$$1! = (1-1)! \cdot 1 = 0! \cdot 1 =^{****} .$$

В останній формулі використовується допоміжне значення 0!, але його, на відміну від попередніх випадків, можна отримати безпосередньо з першого правила:

$$0! = 1,$$

тому, підставивши його, отримаємо

$$1! =^{****}= 1 \cdot 1 = 1.$$

Отримано значення, яке потрібно було, як допоміжне, для обчислення 2!, тому перерване обчислення, помічене трьома зірочками, можна «розморозити» та довести до кінця:

$$2! =^{***}= 1 \cdot 2 = 2.$$

Тепер знайдено те значення, для отримання якого було перервано обчислення 3!. Відновимо його:

$$3! =^{**}= 2 \cdot 3 = 6.$$

Значення 3! колись не вистачало для того, щоб обчислити 4!, тому зараз можна повернутися до першого з перерваних обчислень:

$$4! =^{*}= 6 \cdot 4 = 24.$$

Це і є результат, який треба було знайти.

Приємна особливість рекурентних означень полягає в тому, що їх можна майже автоматично перетворити на програмний текст рекурсивної функції:

```
double factorial( int n ) {
  if( n == 0 )
    return 1;
  return factorial(n-1) * n;
}
```

На с. 64 було наведено іншу реалізацію алгоритму обчислення факторіалу, за допомогою циклу. Реалізація через цикл потребує 8 рядків програмного тексту та додаткових змінних `i` та `p`. Рекурсивна реалізація виявилася на 3 рядки коротшою та не містить змінних для проміжних результатів.

Увага! Практика показує, що і теоретичний матеріал про рекурсивні функції, і задачі на їх застосування часто виявляються для студентів надто важкими. Але особливість полягає саме в тому, що рекурсивні функції, якщо їх один раз добре опрацювати, дозволяють вирішувати своє коло задач значно *простіше*, ніж структурні оператори.

Будь-який алгоритм, який можна виразити з використанням циклів, можливо перетворити в таку еквівалентну форму, де використовується лише рекурсія. Вірно також і обернене твердження: будь-яку рекурсивну програму можна переписати, позбувшися рекурсій, замінивши їх циклами.

Недоліками рекурсивних функцій в мові C є порівняно великі затрати пам'яті та часу. Мова C більше пристосована до операторів циклу. Хоча для багатьох задач рекурсивне рішення виглядає настільки елегантнішим та простішим, що ці додаткові витрати цілком виправдані.

Відзначимо також, що існує ціле сімейство мов, яким загалом не притаманне поняття циклу, натомість використовуються самі лише рекурсії. До них належить одна з найвідоміших мов штучного інтелекту Лісп та вітчизняна розробка — мова Рефал (назва останньої означає «**ре**курсивних **ф**ункцій **ал**горитмічна мова»).

Наостанок можна згадати два програмістські жарти.

- Як прострелити собі ногу рекурсивним методом? — Треба прострелити собі ту кінцівку, якою тримаєте той пістолет, з якого прострелюєте собі кінцівку, якою тримаєте пістолет, з якого прострелюєте собі кінцівку, якою тримаєте пістолет, з якого прострелюєте собі кінцівку...
- Для того, щоб зрозуміти, що таке рекурсія, спершу треба розібратися, що таке рекурсія.

Запитання

1. Що таке рекурсія, для яких задач її варто використовувати?
2. Написати рекурсивну функцію, яка має два аргументи: ціле n та дійсне x та повертає значення x^n.
3. Написати рекурсивну функцію, яка обчислює та повертає n-е число Фібоначчі.

5.7. Підсумковий огляд

Функція — це основна структурна одиниця, з яких складається програма в мові С. Кожна функція має своє ім'я, перелік аргументів певних типів, які мають подаватися їй на вхід, та виробляє в результаті своєї роботи значення певного типу.

Змінні, оголошені всередині функції, тобто локальні змінні, доступні лише для своєї функції, є її внутрішніми деталями реалізації. Вони створюються в момент виклику функції та знищуються при поверненні з неї.

Робота з функцією полягає в тому, що деяка інша функція викликає дану функцію, передаючи до її аргументів значення деяких виразів. Викликана функція виконує всі дії, описані в її тілі, та виробляє певне значення — результат. Цей результат повертається в те місце програми, з якого була викликана функція. Там з ним можна поводитися так само, як і з будь-яким значенням виразу.

Рекурсія — це особливий стиль написання функцій, який добре підходить для програмування багатьох математичних задач. Найтиповіший випадок рекурсії виникає тоді, коли для розв'язання задачі з цілочисельним параметром n треба спершу розв'язати ту ж саму задачу, але для меншого значення параметру, скажімо $n-k$ або $n-1$. До того ж, для деяких значень параметру (наприклад, 0) розв'язок має отримуватися безпосередньо, без потреби розв'язувати допоміжну задачу.

Декомпозиція — розбиття великої і складної програмної системи на сукупність пов'язаних між собою відносно простих підпрограм стала свого часу однією з найвизначніших віх в історії програмування. Зробивши функції основою стилю розробки програм, можна отримати незліченні переваги, в тому числі:

- Програмісту вже не треба тримати в голові одночасно всю програму, з'являється можливість працювати в кожен момент часу над однією невеликою частиною — функцією, що значно спрощує роботу.
- З'являється можливість організувати паралельну роботу кількох програмістів над спільним проектом — оскільки функції відносно незалежні одна від одної, окремі функції можуть розробляти різні люди.
- Функції «все одно», в якій програмі вона використовується та звідки викликається — якщо програма дбає про те, щоб передавати на вхід функції правильні аргументи, то функція буде розв'язує свою чітко окреслену задачу. Отже, ту саму функцію, написавши один раз, можна вставляти в безліч програм. Це дозволяє значно економити зусилля програмістів, оскільки відкидає потребу щоразу писати однаковий код та дозволяє збирати програму з заздалегідь заготовлених функцій.

Розділ 6

Масиви. Макропідстановки

6.1. Макропідстановки

Ключові слова: константа, директива означення макроса `define`, макропідстановка.

Нехай в тексті програми кілька разів використовується одне й те саме число. Наприклад, програма повинна вводити з клавіатури 10 цілих чисел та підраховувати їх середнє арифметичне:

```
1  int main() {
2    int i, x, s=0;
3    double d;
4    for( i = 0; i < 10; ++i ) {
5      printf( "Введіть %d-е число ", i );
6      scanf( "%d", &x );
7      s += x;
8    }
9    /* в s зберігається сума 10 введених чисел */
10   d = (double) s / 10;
11   /* в d зберігається середнє арифметичне */
12   printf( "Середнє %d\n" );
13   return 0;
14 }
```

Тут константа 10 використовується двічі: перший раз як кількість ітерацій циклу, а вдруге — при діленні суми чисел на їх кількість.

Уявімо, що постановка задачі трохи змінюється: знайти середнє арифметичне 15 чисел. Виникає потреба замінити в тексті програми константу 10 на константу 15. Іншими словами, програмісту треба самостійно продивитися весь текст, знайти в ньому всі входження однієї константи та вписати замість них іншу. Якщо текст програми великий та складний, то ця марудна та механічна робота забирає надто багато сил.

При цьому легко помилитися, випадково пропустивши одну з таких замін. В даному прикладі програміст може замінити, як вимагається, першу десятку (в циклі), але забути про другу (при діленні). Програма буде обраховувати неправильний результат, причому помітити таку помилку у великій програмі дуже важко.

Крім того, незрозумілим з тексту програми залишається сенс константи. Саме по собі записане число — безглузде. Стороння людина, читаючи програму[1], не одразу зрозуміє, що означає те чи інше число. В наведеному вище прикладі з тексту програми не зрозуміло, що мається на увазі: 10 чисел чи 10 вольт, чи 10 метрів, чи 10 байт пам'яті.

Ще серйозніша проблема виникає, коли в одній програмі використовується кілька констант з різним предметним змістом, але з однаковими числовими значеннями. Наприклад, в тій же програмі може бути друк 10 порожніх рядків на початку роботи програми:

```
...
for( i = 0; i < 10; ++i )
  printf( "\n" );
...
```

[1]Завжди треба зважати на те, що програма пишеться не тільки для того, щоб її виконувала машина, але й для того, щоб її читала людина. Текст програми це не лише інструкція для комп'ютера, але й засіб обміну знаннями між людьми.

Цілком очевидно, що ця десятка за своїм змістом та призначенням зовсім відрізняється від десятки, обведеної рамкою. Зокрема, при зміні постановки задачі (знайти середнє арифметичне 15 чисел) обведені рамкою десятки треба замінити на 15, а необведену — ні. Але ж по вигляду вони не відрізняються, і програмісту важко вирішити, яку з них замінити, а яку ні. Це створює небезпеку помилки.

Всі зазначені проблеми легко вирішуються за допомогою директиви означення макросу. Проілюструємо її спочатку прикладом:

```
1  /*це означення констант*/
2  #define N_VALUES 10 /*чисел для обробки*/
3  #define N_BLANK_LINES 10 /*порожніх рядків*/
4
5  int main() {
6    int i, x, s=0;
7    double d;
8    for( i = 0; i < N_BLANK_LINES; ++i )
9      printf( "\n" );
10   for( i = 0; i < N_VALUES; ++i ) {
11     printf( "Введіть␣%d-е␣число␣", i );
12     scanf( "%d", &x );
13     s += x;
14   }
15   /* тепер в змінній s зберігається
16      сума 10 введених чисел */
17   d = (double) s / N_VALUES;
18   /* тепер в змінній d зберігається
19      середнє арифметичне*/
20   printf( "Середнє␣%d\n" );
21   return 0;
22 }
```

Тут за допомогою *директиви означення макросу* **#define** означується константа з іменем `N_VALUES` та значенням `10`. Відтепер всюди в тексті програми після цієї директиви транслятор, «побачивши» ідентифікатор `N_VALUES`, замінить його на текст `10`. Аналогічно оброблюється означення другої константи `N_BLANK_LINES`.

Вигоди від використання таких констант очевидні. По-перше, при зміні постановки задачі (обчислити середнє арифметичне 15 чисел) достатньо замінити лише в одному місці означення константи `N_VALUES`, а транслятор сам підставить нове значення всюди, де потрібно. При цьому програмісту не потрібно довго шукати в тексті програми, де треба зробити заміну, оскільки за традицією директиви означення макросів пишуть на самому початку файлу.

По-друге, символічне ім'я одразу робить текст програми прозорим та зрозумілим: побачивши в тексті програми запис `N_VALUES`, сторонній читач зрозуміє[1], яка роль цієї константи в програмі.

Отже, в простому випадку[2] означення макросу має вигляд

```
#define ідентифікатор текст
```

з будь-яким ідентифікатором та будь-яким текстом. В подальшому тексті програми транслятор замінює всі входження даного `ідентифікатору` на даний `текст`

Увага! В лабораторних роботах обов'язково використовувати для всіх констант символічні імена. Числа в тілі програми забороняються, крім небагатьох очевидних винятків (наприклад, 0 як початкове значення суми). Студентам треба з самого початку привчитися писати програми гарно.

[1]Якщо розуміє англійську. Знання англійської мови абсолютно необхідне програмісту-професіоналу, оскільки у всіх розповсюджених мовах програмування ключові слова, імена функцій, констант тощо взяті з англійської.

[2]Існує також більш складний випадок макросів з параметрами, який розглядати не будемо.

Означення констант — важливе, але не єдине можливе призначення директиви **#define**. Ще один спосіб її використання буде описано в розділі 12.3.

Запитання

1. Написати програму, що переводить кути з радіанів в градуси, означивши число π (3,1415926) та відповідне число градусів (180) за допомогою макросів.
2. Написати програму, яка розраховує час падіння тіла з відомої висоти. Константу **g** (прискорення вільного падіння) означити за допомогою макросу.
3. Написати програму, яка виводить на екран прямокутник із зірочок, ширина та висота якого задані за допомогою макросу.

6.2. Поняття масиву

Ключові слова: масив, розмір, індекс, операція індексування, звертання до елементів масиву, ініціалізація елементів масиву.

В усіх програмах, що розглядалися вище, оброблялися, так би мовити, поодинокі значення. На практиці, однак, часто виникає потреба обробити єдиним алгоритмом велику сукупність однорідних значень. В математиці такі сукупності мають позначення на зразок $x_1, x_2, \ldots, x_n$. Тут літерою x позначається вся сукупність чисел, як єдине ціле, а через x_i позначається її окремий елемент, i-й за номером від початку сукупності.

Нехай, наприклад, на насосній станції кожну годину вимірюється тиск у трубопроводі. Тоді результати вимірювання за добу утворюють сукупність $p_1, p_2, \ldots, p_{24}$. Типовими прикладами обробки такої сукупності значень є: обчислення середнього арифметичного чи середнього геометричного, пошук найбільшого або найменшого елементу, сортування сукупності (тобто така перестановка елементів, щоб вони розташувалися у порядку збільшення чи зменшення).

Наведені міркування стосувалися одновимірних сукупностей, але ніщо не заважає, якщо це потрібно за умовою задачі, використовувати і двовимірні сукупності — матриці $M = [m_{i\,j}]$, а також три- та більшого числа вимірів.

Для підтримки обробки таких сукупностей в мові C існує поняття *масиву*. Масив являє собою особливий різновид структур даних — послідовність з певної кількості перенумерованих елементів однакового типу. В цьому розділі будемо розглядати одновимірні масиви, кількість елементів у яких жорстко визначено заздалегідь. В загальному випадку оголошення такого масиву має вигляд:

```
ім'я_типу ім'я_масиву[ константа_розмір ];
```

Це оголошення відрізняєтсья від оголошення звичайної «одиночної» змінної наявністю розміру, записаного у квадратних дужках. Треба звернути увагу, що розмір, який вказується при оголошенні масиву, повинен бути константою, тобто не може бути значенням змінної чи, скажімо, результатом виклику функції. Наприклад, оголошення масивів з 12 цілих чисел та з 17 дійсних має вигляд

```
int trail[ 10 ];
double value[ 17 ];
```

Увага! Нумерація елементів в масиві починається з 0. Якщо масив містить N елементів, то це значить, що в ньому є елементи під номерами 0, 1, ..., $N-1$. Елемента номер N в масиві немає.

Для доступу до елементу масиву слугує операція *індексування*, яка позначається квадратними дужками: конструкція `ім'я_масиву[номер_елементу]` означає змінну — елемент, що стоїть в даному масиві під даним номером. Номер елементу в масиві називають також *індексом*. Звичайно ж, номер елементу може бути заданий будь-яким виразом цілого типу. Звертання до елементу масиву нічим в принципі не відрізняється від звертання до звичайної змінної за іменем, хіба що в даному випадку в ролі імені змінної виступає, так би мовити,

складне ім’я, що складається з імені самого масиву та індексу елементу. Наприклад, розглянемо програмний фрагмент:

```
1 int m[ 10 ], k = 3;
2 m[0] = 1;
3 m[k] = 8;
4 ++k;
5 m[k] = 8;
6 m[(k+2)%3+1] = 17;
7 m[k+3] = m[0]+m[k];
8 scanf( "%d", &m[k+1] );
9 printf( "%d\n", m[k] );
```

В першому рядку оголошується масив `m` з 10 елементів та допоміжна змінна `k`, яка одразу отримує початкове значення 3. В рядку 2 показано, як присвоїти значення елементу масиву, номер якого заздалегідь відомий: в якості індексу використано константу, число 0. Оскільки нумерація елементів починається з 0, то даний оператор означає, що значення присвоюється першому елементу масиву.

Рядок 3 ілюструє, що індекс може бути не константою, а значенням змінної. Оскільки в даний момент змінна `k` має значення 3, даний оператор означає, що значення 8 присвоюється у четвертий від початку (а не третій!) елемент масиву. Оператор в рядку 4 збільшує значення змінної `k` на 1, отже, воно тепер дорівнює 4. Тому, хоча оператор в рядку 5 повністю співпадає за написанням з оператором в рядку 3, тепер вираз в лівій частині присвоювання означає вже не четвертий, а п’ятий від початку елемент масиву.

Рядок 6 є прикладом того, що в якості індексу може використовуватися не лише значення змінної, але і як завгодно складний вираз. Підставивши поточне значення змінної `k`, маємо, що значення 17 буде присвоєно елементові з індексом 1, тобто другому елементу масиву.

В рядку 7 показано, що звертання до елементів одного й того самого масиву може здійснюватися і в лівій, і в правій частинах присвоювання. В перший (з індексом 0) елемент раніше було занесено значення 1, поточне значення змінної `k` дорівнює 4, а елементу з індексом 4 було присвоєно значення 8. Отже, елемент з індексом 7 (восьмий від початку) отримає значення 9.

Значення елементів масиву можна вводити з клавіатури так само, як і значення звичайних змінних, за допомогою функції `scanf`, що показано в рядку 8. Як і завжди, перед іменем змінної, в яку треба розмістити введене значення, ставиться знак `&` — амперсанд.

З останнього рядка прикладу видно, що значення елементів масиву можна передавати до функцій в якості аргументів, в тому числі — друкувати на екран.

Одразу ж при оголошенні масиву можна присвоювати значення його елементам, або, як кажуть, *ініціалізувати* масив. Для цього достатньо записати знак рівності та послідовність значень елементів в фігурних дужках:

```
int m[4] = { -1, 3, 0, 27 };
```

Більш того, в такому випадку розмір масиву можна не вказувати:

```
int m[] = { -1, 3, 0, 27 };
```

Тоді транслятор сам визначить розмір, порахувавши ініціалізатори.

Підкреслимо, що присвоїти масиву одразу цілу сукупність значень можна лише при оголошенні масиву, тобто задавши початкові значення його елементам. По ходу виконання програми, серед операторів, присвоїти один масив іншому масиву (так, щоби з одного масиву в інший скопіювалися значення одразу всіх елементів) неможливо — див. далі.

Запитання

1. Що таке масив? Для яких задач використовуються масиви? Приклад математичного поняття, яке зручно відобразити в програмі за допомогою масиву.
2. Оголошення масиву в мові C. Як нумеруються елементи масиву?

3. Як виглядає звертання до елементу масиву в мові C?
4. Як надати елементам масиву початкові значення?

6.3. Типова схема обробки масивів

Ключові слова: поточний елемент масиву, поелементний перебір масиву.

В розібраному вище прикладі кожен оператор працював з одним елементом масиву. Та найбільша вигода від масивів, їх гнучкість та потужність проявляються там, де потрібно обробити весь масив як ціле. Яким би не був зміст цієї обробки, у величезній кількості задач для цього використовується ідея послідовної обробки, елемент за елементом. В кожен момент часу алгоритм продивляється один елемент масиву — *поточний елемент*. На початку поточним робимо перший елемент масиву. Далі на кожній ітерації циклу поточний елемент обробляється, після чого алгоритм переходить до наступного елементу, тобто наступний елемент стає поточним. Процес закінчується, коли всі елементи масиву вичерпано. Розпишемо загальну схему алгоритму в деталях.

1. Взяти поточним перший елемент масиву.
2. Масив вже закінчився?
   - Якщо так, то закінчити;
   - Якщо ні, то продовжувати з наступного кроку.
3. Обробити поточний елемент.
4. Взяти поточним наступний елемент масиву.
5. Перейти до кроку 2.

Нехай масив має ім'я `m` та містить N елементів. Щоб виділити в масиві поточний елемент, зрозуміло, потрібно використати цілочисельну змінну `k` — номер (індекс) поточного елементу. Тоді сам поточний елемент є `m[k]`. Фраза «взяти поточним перший елемент масиву» зі словесного опису алгоритму уточнюється в мові C оператором `k=0`, а фраза «взяти поточним наступний елемент масиву» — оператором `++k`. Умова «масив ще не закінчився» (умова продовження циклу) природно уточнюється виразом `k<N` (нагадаємо, що оскільки нумерація елементів масиву починається з 0, то останній елемент має номер $k-1<N$; якщо ж номер поточного елементу після чергового збільшення досягає значення N, то це означає, що всі елементи масиву вже перебрано).

Отже, основна *ідіома*[1] обробки масивів виглядає так:

```
for( k = 0; k < N; ++k )
  обробити m[k];
```

Наприклад, наведений нижче програмний фрагмент вводить масив дійсних чисел елемент за елементом.

```
. . .
#define N 10
int main() {
double m[ N ];
int k;
for( k = 0; k < N; ++k ) {
  printf( "Значення номер %d ", k );
  scanf( "%ld", &m[k] );
}
. . .
```

[1] Ідіомою в програмуванні називають певний спосіб використання кількох операторів для вирішення типової задачі, тобто заготовку програмного коду, яка вирішує якусь типову задачу, та яку можна підставляти у різноманітні програми.

В наведеному прикладі проілюстровано також ще одну важливу деталь. З точки зору гарного професійного стилю, константу «кількість елементів масиву» абсолютно необхідно оформлювати як макрос (див. розділ 6.1). Справді, якби в тексті програми всюди, де потрібна кількість елементів, було жорстко вписане конкретне число 10, то при зміні умови задачі (скажімо, замість 10 ввести 12 чисел), було б дуже важко внести зміну в усіх місцях, де потрібно.

Увага! Використання макроозначень для розмірів масивів — обов'язкова ознака професійного підходу до програмування. Лабораторні роботи, написані без додержання цієї вимоги, не приймаються.

6.4. Обчислення значень поліному

Розберемо принцип поелементної обробки масивів на прикладі задачі: обчислити значення заданого поліному P_n для довільного x. Поліном (багаточлен) n-го ступеню — це функція вигляду

$$P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = \sum_{i=0}^{n} a_i x^i.$$

Очевидно, для того, щоб задати поліном, достатньо задати сукупність чисел $a_0, a_1, \ldots, a_n$ (коефіцієнтів). Тому поліном n-го ступеню в програмі найзручніше змоделювати за допомогою масиву, що містить $n+1$ елементів дійсного типу, значеннями яких є відповідні коефіцієнти.

Отже, найпростіша програма, яка б розв'язувала поставлену задачу (при $n = 5$), мала б таку будову.

```
1  #include <stdio.h>
2  #define N 5 /* степінь поліному */
3  int main() {
4    double s, t, a[ N+1 ];
5    int i;
6    ввести_коефіцієнти;
7    ввести_значення_x;
8    s = 0; /* накопичувач суми */
9    for( i = 0; i < N+1; ++i ) {
10     t = (обчислити x^i);
11     s += t * a[i];
12   }
13   друкувати_значення_s;
14   return 0;
15 }
```

Довести цю заготовку до завершеного вигляду не становить труднощів (див. завдання наприкінці розділу). Але така програма має суттєву ваду, оскільки витрачає багато часу на зайві обчислення, отже неефективно використовує ресурс процесору. Справді, розглянемо останню ітерацію циклу, коли змінна i вже досягла значення n. В тілі циклу потрібно обчислити значення x^n, а це означає помножити значення x саме на себе n разів. В процесі такого обчислення будуть з'являтися такі проміжні значення, як x^2, x^3, x^{n-1}. Всі ці проміжні значення вже обчислювалися на попередніх ітераціях циклу, але після обчислення одразу «забувалися», і тепер програмі доводиться обчислювати їх знову.

На кожній ітерації робиться одна операція додавання та i операцій множення, тому в загальному підсумку обчислення поліному потребує $\frac{n^2-n}{2}$ множень та n додавань, тобто при збільшенні степеню поліному витрати процесорного часу зростають пропорційно квадрату n.

Отже, було б бажано значення x^i на кожній ітерації не «викидати», а зберігати та використовувати на наступній ітерації для обчислення наступного степеню числа x. З наведених міркувань отримуємо більш економний варіант програми (наводимо лише ту частину, яка зазнала змін):

```
. . .
s = 0; /* накопичувач суми */
t = 1; /* поточний степінь x */
for( i = 0; i < N+1; ++i ) {
  s += t * a[i];
  t *= x;
}
```

Тут на кожній ітерації виконується два множення та одне додавання, отже при збільшенні ступеню поліному витрати часу зростають пропорційно n (лінійно).

Але ця програма також не найекономніша: виявляється, що можливо скоротити кількість множень до одного на кожну ітерацію. Для цього потрібно скористатися так званою схемою Горнера. Розглянемо поліном P_n як суму двох доданків, перший з яких є складається з усіх членів, що містять x, а другий є числом a_0 (член, який не містить x). У першому доданку винесемо x за дужки.

$$P_n(x) = \left(a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x\right) + a_0 = \\ \left(a_n x^{n-1} + a_{n-1}x^{n-2} + \ldots + a_1\right) x + a_0.$$

В першому доданку в дужках залишився вираз, який також є поліномом, тільки вже степеню $n-1$. З ним зробимо так само: відокремимо останній доданок, а з суми решти членів винесемо за дужки x, і так далі. Отримаємо вираз

$$P_n(x) = (\ldots((a_n x + a_{n-1})\, x + a_{n-2})\, x + a_{n-3}\ldots)\, x + a_0.$$

або, у рекурентній формі,

$$P_0(x) = a_n,$$

$$P_{i+1}(x) = P_i(x) \cdot x + a_{n-i}.$$

Сенс рекурентного означення такий: на першому кроці взяти значення коефіцієнту a_n; на кожному наступному кроці брати значення, отримене на попередньому, множити його на x та додавати наступний (в порядку спадання) коефіцієнт. Тоді через n кроків буде отримано значення поліному, яке потрібно було знайти.

Програмний текст мовою C, що обчислює значення поліному за схемою Горнера, наведено нижче:

```
1  #include<stdio.h>
2  #define N 5 /* степінь поліному */
3  int main() {
4    double s, a[ N+1 ];
5    int i;
6    /* ввести коефіцієнти */
7    printf( "Введіть␣%d␣коефіцієнтів", N+1 );
8    for( i = 0; i < N+1; ++i )
9      scanf( "%lf", &a[i] );
10   scanf( "%lf", &x );
11   s = 0; /* накопичувач суми */
12   for( i = N; i >= 0; i-- ) {
13     s *= x;
14     s += a[i];
15   }
16   printf( "P(%lf)␣=␣%lf\n", x, s );
17   return 0;
18 }
```

Звернімо увагу, що в цій програмі елементи масиву обробляються у зворотному порядку, від кінця до початку.

Запитання

Довести до програмної реалізації перший та другий варіанти програми обчислення значень поліному. Вказівка: у першому варіанті винести обчислення x^i в окрему функцію.

6.5. Лінійний пошук

Ключові слова: задача пошуку в масиві, перебір всіх елементів.

Серед найтиповіших задач, які на практиці часто трапляються в складі більш об'ємних задач, важливе місце посідають задачі *пошуку* в масиві. Нехай дано масив m з N елементів та число x, яке називають *ключем*. Треба знайти номер i такого елементу в масиві, який дорівнює ключу: $m_i = x$, або встановити, що такого елементу в масиві немає, тобто ключ не має входжень в масив.

Найпростіший та найуніверсальніший (але водночас і найгірший за часом роботи) алгоритм полягає в тому, щоб продивлятися кожен по черзі елемент масиву m, та перевіряти, чи не співпадає він з ключем. В разі, якщо співпадає, перегляд послідовності елементів завершується. Розпишемо цей алгоритм в подробицях:

1. Взяти поточним перший елемент масиву.
2. Масив вже закінчився?
   - Якщо так, то пошук закінчується невдачею: потрібного значення в масиві немає;
   - Якщо ні, то продовжувати з наступного кроку.
3. Поточний елемент співпадає з ключем?
   - Якщо так, то пошук закінчено, потрібне значення знайдене;
   - Якщо ні, продовжувати з наступного кроку.
4. Взяти поточним наступний елемент масиву.
5. Перейти до кроку 2.

Видно, що обробка поточного елементу в тілі циклу передбачає дострокове (до досягнення кінця масиву) завершення циклу. Це досягається за допомогою оператору **break** (див. розділ 3.5). Якщо елемент, який дорівнює ключу, в масиві знайдено, то після дострокового переривання циклу змінна k (номер поточного елементу) буде обов'язково мати значення з проміжку від 0 до $N - 1$ включно (номер знайденого елементу масиву). Якщо ж ключа в масиві не знайдено, то після завершення циклу змінна k буде мати значення N (яке вже не є номером якогось елементу масиву). За допомогою цих міркувань приходимо до такого фрагменту програми (для спрощення пропущено частину, яка вводить елементи масиву).

```
1  #include<stdio.h>
2  #define N 20
3  int main() {
4    int m[N], x, k;
5    /* пропущено введення масиву */
6    printf( "Що␣шукати?␣" );
7    scanf( "%d", &x );
8    for( k = 0; k < N; ++k )
9      if( m[k] == x )
10       break;
11   if( k < N )
12     printf( "Знайдено␣під␣номером␣%d\n", k );
13   else
14     printf( "Такого␣числа␣в␣масиві␣немає\n" );
15   return 0;
16 }
```

Якщо в масиві є декілька входжень ключа (тобто кілька елементів мають значення x), то оскільки алгоритм передивляється всі елементи масиву від початку до кінця, він знайде *перший* з цих елементів (див. вправи). Алгоритм лінійного пошуку дозволяє шукати не лише входження певного значення ключа, але й шукати елемент, який задовольняє більш складній умові (див. вправу).

Запитання

1. Перетворити програму лінійного пошуку так, щоб вона шукала в масиві не перше, а останнє входження ключа.
2. Перетворити програму лінійного пошуку так, щоб вона шукала в масиві всі входження ключа і друкувала на екран номери всіх знайдених елементів.
3. Перетворити програму лінійного пошуку так, щоб вона шукала в масиві перше число, яке ділиться на 3 і одночасно більше за 25.

6.6. Сортування: метод бульбашки

Ключові слова: задача сортування, впорядкований масив.

Дуже важливу область задач, пов'язаних з масивами, становлять задачі *сортування*: дано масив, алгоритм повинен так переставити його елементи, щоб вони розташувалися в порядку зростання (або спадання).

Розглянемо найпростіший за своєю ідеєю та принципом дії, але дуже повільний алгоритм *бульбашки*. Його основна ідея полягає в наступному. Будемо продивлятися масив елемент за елементом від початку. На кожному кроці дивимося лише на два сусідні елементи: поточний та наступний за ним. Може статися, що пара елементів, які ми в даний момент продивляємося, стоять у «правильному» порядку зростання: наступний більше за поточний. В такому випадку можна просунутися далі до наступного елементу. В іншому випадку пару елементів треба поміняти місцями, щоб вони стали в правильному порядку, «перестрибнути» на початок масиву і почати весь процес знову.

Розглянемо приклад роботи алгоритму, рис. 6.1. Пару елементів «поточний–наступний» позначено подвійною рамкою. На першому кроці алгоритму розглядається пара, що складається з першого та другого елементів масиву. Порядок елементів в парі правильний, тому алгоритм йде по масиву далі (показано стрілкою). На другому кроці розглядається пара з другого та третього елементів. Ця пара порушує порядок сортування, тому алгоритм міняє ці елементи місцями. Після цього алгоритм повинен перейти до початку масиву і на кроці 3 розглянути знов пару з першого та другого елементів. Оскільки ця пара стоїть в правильному порядку, алгоритм просувається до наступної пари. На кроці 4 розглядаються другий та третій елементи. Вони також стоять в правильному порядку, тому алгоритм просувається ще на один елемент. На кроці 5 маємо пару чисел 28 і 7, яка порушує порядок зростання (друге число менше за перше). Тому алгоритм міняє їх місцями і повертається на початок масиву.

Крок 6 алгоритму обробляє пару з першого та другого елементів масиву, які стоять у порядку зростання, тому поточним стає наступний елемент. На кроці 7 пара з другого та третього елементів стоїть у протилежному порядку, тому елементи міняються місцями і алгоритм знов починає обробку масиву спочатку. Поглянувши на масив, що утворився на кроці 8, можна переконатися, що всі його елементи вже стоять в потрібному порядку зростання, але цей факт, очевидний для людини, зовсім не очевидний для машини, і алгоритму ще належить його перевірити. На кроках 8, 9 та 10 алгоритм бачить, що поточна пара елементів стоїть у правильному порядку, та переходить до наступної пари. Після кроку 10 алгоритм досягає кінця масиву і тим самим переконується, що всі його елементи впорядковано за зростанням, а отже завершує свою роботу.

Звернімо увагу на умову зупинки алгоритму. Щоразу алгоритм починає передивлятися весь масив з самого початку і продивляється його до першого порушення порядку. Отже,

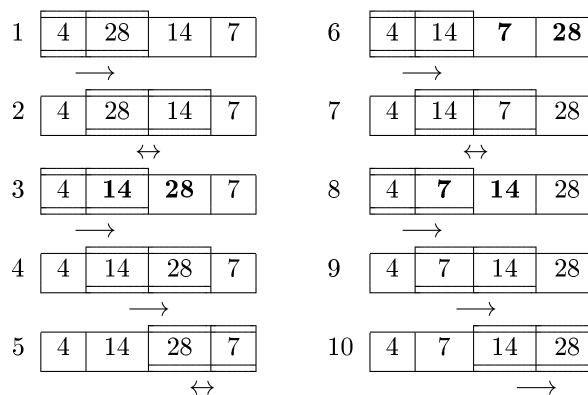

Рис. 6.1. Сортування масиву за методом бульбашки

якщо на деякому кроці поточним є елемент номер i, то за логікою роботи алгоритму *гарантується*, що всі елементи від початку масиву до i-го включно стоять у порядку зростання. Таким чином, якщо алгоритму вдасться дійти до кінця масиву, буде гарантовано, що впорядковано весь масив.

Після такого детального розгляду буде неважко зрозуміти програмний текст. Щоб не перевантажувати програму зайвими подробицями, директиви підключення заголовків, введення початкових даних та виведення результату не наводимо.

```
1  #define N 10
2  int main() {
3    int i;
4    double m[ N ], t;
5    /* введення даних пропущено */
6    i = 0;
7    while( i < N-1 ) {
8      if( m[i+1] >= m[i] )
9        ++i;
10     else {
11       t = m[i];
12       m[i] = m[i+1];
13       m[i+1] = t;
14       i = 0;
15     }
16   }
17   /* друк результату пропущено */
18   return 0;
19 }
```

Запитання

1. Відсортувати «вручну» методом бульбашки (див. приклад) послідовність чисел: 13, 19, 5, 16, 11.
2. Запустити наведену вище програму сортування масивів, переконатися, що для різних вхідних даних вона працює правильно.
3. В тексті програми стоїть оператор `if( m[i+1] >= m[i] )`. Чому в умові застосовано порівняння «більше або дорівнює», а не «строго більше»? Як змінилася б поведінка алгоритму, якщо б було порівняння «строго більше»?
4. Алгоритм бульбашкового сортування працює дуже повільно для великих масивів, оскільки кожен раз після перестановки двох елементів багато часу витрачається на

зайві просмотри початку масиву. Вдосконалити алгоритм таким чином: на кожній ітерації продивлятися весь масив та робити в ньому всі можливі перестановки пар сусідніх елементів; якщо хоча б одну перестановку було зроблено, повторити процес. Скласти програму, яка реалізує вдосконалений алгоритм, переконатися, що вона працює.

6.7. Пошук поділом навпіл

Повернімося ще раз до задачі пошуку в масиві і з'ясуємо, чи можливо проводити пошук значно швидше, ніж переглядом кожного елементу. На прикладі словників, телефонних довідників тощо легко переконатися в користі від впорядкування елементів. Справді, відшукати в телефонному довіднику потрібне прізвище серед тисяч записів було б майже неможливо, якби прізвища йшли в довільному порядку. А завдяки впорядкуванню за абеткою такий пошук здійснюється досить швидко.

Припустімо, що масив m, що складається з N елементів відсортовано за зростанням; це означає, що $m_i \leqslant m_j$ при $i < j$. Нехай потрібно знайти в масиві значення x. Поділимо масив навпіл: нехай число k є номером елементу, що стоїть посередині масиву, ліва половина розташована від початку масиву до $k-1$-го елементу, права половина складається з елементів від $k+1$-го до кінця масиву.

Може статися так, що $m_k = x$ — це означає, що потрібне значення вже знайдено. Якщо ж $m_k \neq x$, то можливі два випадки: $x < m_k$ та $x > m_k$. В першому з цих випадків очевидно, що значення x може входити лише в ліву половину масиву: справді, оскільки x менше за k-й елемент, а масив впорядковано за зростанням, то напевне x буде менше за будь-який елемент, що йде після k-го. Аналогічним чином переконуємося, що у випадку $x > m_k$ можна одразу відкинути з розгляду всю ліву половину масиву, бо в ній завідомо немає сенсу шукати x.

В обох випадках область подальшого пошуку звузилася наполовину. Для того, щоб тепер знайти ключ в половині, що залишилася, скористаємося ще раз таким же поділом навпіл, і так далі до тих пір, поки після чергового поділу не натрапимо на елемент, який шукали. Може також статися, що після чергового поділу область пошуку виявиться порожньою: половина масиву, що залишилася, не містить жодного елементу. Це означає, що в масиві немає жодного елементу, значення якого дорівнює x.

Нехай, наприклад, дано впорядковану послідовність з 31 числа: 12, 30, 63, 65, 68, 69, 116, 122, 127, 128, 130, 137, 173, 178, 183, 193, 224, 227, 231, 236, 236, 241, 250, 260, 264, 268, 284, 287, 287, 291, 294, і нехай потрібно знайти, на якому місці в ній стоїть число $x = 241$. На першому кроці область можливого пошуку (та частина масиву, в якій ми можемо сподіватися знайти ключ) охоплює весь масив. Серединою масиву є елемент під номером $k = 15$, його значення $m_k = 193$. Оскільки $x > m_k$, можна відкинути всі елементи з 0-го по 15-й включно, тепер область пошуку складається з елементів з 16-го по 30-й.

Серединою цієї області є елемент під номером $k = 23$, $m_k = 260$. Порівняння дає $x < m_k$, тому половина області пошуку, елементи з 23-го по 30-й відкидаються, для пошуку залишається область з 7 елементів, з 16-го по 22-й. Її серединою є елемент під номером $k = 19$, $m_k = 236$. Оскільки $x > m_k$, подальший пошук проводиться серед елементів з 20-го по 22-й. Серединою цієї області є елемент $m_{21} = 241 = x$, отже ключ знайдено, відповідь задачі — $k = 21$.

Для того, щоб втілити описану загальну ідею в алгоритмі, знадобляться змінні l та r, які обмежують область пошуку: від l-го елементу до $r-1$-го. Пошук є сенс продовжувати доти, доки область пошуку не порожня, містить хоча б один елемент — а це виражається співвідношенням $l < r$. На кожному кроці або права, або ліва границя області пошуку пересувається на те місце, де раніше була середина області — це реалізується присвоєнням змінній l або r відповідного значення.

Алгоритм може закінчити роботу двома шляхами: або коли знайдено потрібний елемент, або коли область пошуку стала порожньою. Тому після завершення циклу необхідно перевірити, чи справді індекс k вказує на елемент, який співпадає з ключем. В підсумку маємо такий програмний текст (самоочевидний фрагмент на початку програми пропущено).

```
#include <stdio.h>
#define N 20
int main() {
  int m[ N ], l, r, k, x;
  /* пропущено введення та сортування */
  printf( "Яке значення шукати? " );
  scanf( "%d", &x );
  l = 0;
  r = N;
  while( l < r ) {
    k = l + (r-l) / 2;
    if( m[k] == x ) break;
    if( m[k] < x ) l = k + 1;
    else r = k;
  }
  if( m[k] == x )
    printf( "Знайдено під номером %d", k );
  else
    printf( "Не знайдено\n" );
  return 0;
}
```

Звернімо увагу, що для визначення середини області пошуку застосовано формулу `l+(r-l)/2` замість більш звичного `(l+r)/2`. Справа в тому, що якщо масив виявиться надзвичайно великим, а саме, коли його довжина близька до найбільшого числа типу **`int`**, додавання величин `l` та `r` може призвести до переповнення розрядної сітки. Формула, застосована в даному прикладі, гарантовано не призведе до такої помилки.

На завершення розгляду порівняємо алгоритми лінійного пошуку та поділу навпіл. В алгоритмі лінійного пошуку область, де можна ще сподіватися знайти потрібне значення, з кожним кроком зменшується на один елемент, а в алгоритмі поділу навпіл — зменшується одразу наполовину. Тому в середньому час роботи алгоритму лінійного пошуку пропорційний довжині масиву: якщо, наприклад, масив збільшиться вдвічі, то в стільки ж разів зросте час пошуку. А алгоритм поділу навпіл при збільшенні довжини масиву вдвічі потребує лише одну додаткову операцію. Взагалі, якщо довжина масиву становить n, то час роботи цього алгоритму пропорційний $\log_2 n$.

Водночас, в алгоритмі лінійного пошуку масив може бути будь-яким, жодні вимоги щодо його особливих властивостей не накладаються. Тому цей алгоритм є найуніверсальнішим. Натомість, в алгоритмі пошуку поділом навпіл є суттєве обмеження: вимагається, щоб масив був впорядкований. Зокрема, якщо початкові дані надходять невпорядкованими, то перед застосуванням цього алгоритму потрібно витратити додаткові зусилля, щоб виконати алгоритм сортування.

6.8. Прості числа, решето Ератосфена

Проілюструємо використання масивів це одним цікавим прикладом. Нехай задане деяке достатньо велике ціле число N та треба знайти всі прості числа від 1 до $N-1$. Цю задачу можливо розв'язувати безліччю різноманітних методів, але зараз розглянемо один, що має назву «решето Ератосфена».

Випишемо підряд числа від 0 до $N-1$. На кожному кроці алгоритму в цій послідовності мають бути підкреслені ті числа, щодо яких вже відомо, що вони прості; числа, серед яких немає сенсу шукати прості, бо вони завідомо діляться на раніше знайдені прості, закреслено. Тому потрібно знайти в послідовності перше зліва число, яке ще не викреслено і не підкреслено. Знайдене число є простим (справді, оскільки його не викреслено на жодному з попередніх кроків, то воно не ділиться на жодне з попередніх чисел і отже є простим), тому його треба підкреслити, а далі закреслити всі числа, кратні йому.

Наведемо декілька кроків роботи алгоритму при $N = 16$. Числа 0 та 1, як відомо, не є простими, тому їх одразу викреслюємо. Першим простим числом є 2, тому число 2 підкреслимо, а всі числа, кратні 2, закреслимо. Далі знайдемо в послідовності перше (найлівіше) незакреслене число — це буде 3. Підкреслимо його як просте і закреслимо всі числа, кратні 3. Далі шукаємо в послідовності наступне найлівіше незакреслене число, ним виявиться 5, і проведемо з ним ту ж процедуру.

~~0~~	~~1~~	$\underline{\mathbf{2}}$	3	~~4~~	5	~~6~~	7	~~8~~	9	~~10~~	11	~~12~~	13	~~14~~	15
~~0~~	~~1~~	$\underline{2}$	$\underline{\mathbf{3}}$	~~4~~	5	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~
~~0~~	~~1~~	$\underline{2}$	$\underline{3}$	~~4~~	$\underline{\mathbf{5}}$	~~6~~	7	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~
~~0~~	~~1~~	$\underline{2}$	$\underline{3}$	~~4~~	$\underline{5}$	~~6~~	$\underline{\mathbf{7}}$	~~8~~	~~9~~	~~10~~	11	~~12~~	13	~~14~~	~~15~~
~~0~~	~~1~~	$\underline{2}$	$\underline{3}$	~~4~~	$\underline{5}$	~~6~~	$\underline{7}$	~~8~~	~~9~~	~~10~~	$\underline{\mathbf{11}}$	~~12~~	13	~~14~~	~~15~~
~~0~~	~~1~~	$\underline{2}$	$\underline{3}$	~~4~~	$\underline{5}$	~~6~~	$\underline{7}$	~~8~~	~~9~~	~~10~~	$\underline{11}$	~~12~~	$\underline{\mathbf{13}}$	~~14~~	~~15~~

Алгоритм зупиняє свою роботу, коли в послідовності всі числа будуть підкреслені або закреслені (тобто коли не залишиться неперевірених кандидатів, щодо яких невідомо, є вони простими чи ні).

Потрібно обрати спосіб, щоб зобразити в програмі послідовність чисел, елементи якої можна було б закреслювати та підкреслювати. Скористаємося масивом цілих чисел з таким сенсом: масив p має N елементів. Значення p_i зображує *стан* числа i в Ератосфеновому решеті: p_i дорівнює 0, якщо число ще не розглянуте (не підкреслене та не закреслене); 1 для підкресленого числа та 2 для закресленого.

Наприклад, перед початком роботи алгоритму закреслено числа 0 та 1, а всі інші числа в послідовності «чисті», отже в перших двох елементах (з індексами 0 та 1) стоїть значення 2, а в усіх інших елементах масиву мають стояти нулі. Після першого кроку підкреслено число 2, викреслено всі інші парні числа, це зображується в масиві таким чином:

2	2	1	0	2	0	2	0	2	0	2	0	2	0	2	0

Ще раз звернімо увагу: в масиві зберігаються не самі числа з послідовності, що розглядається (ці числа $2, 3, \ldots, N$ заздалегідь відомі і не змінюються в процесі обробки), а їх підкресленість чи закресленість. На подальших кроках, відповідно, значеннями елементів масиву будуть:

2	2	1	0	2	0	2	0	2	0	2	0	2	0	2	0
2	2	1	1	2	0	2	0	2	2	2	0	2	0	2	2
2	2	1	1	2	1	2	0	2	2	2	0	2	0	2	2
2	2	1	1	2	1	2	1	2	2	2	0	2	0	2	2

Тепер можливо записати алгоритм «решето Ератосфена» у вигляді покрокового словесного опису.

1. Заповнити масив m нулями, в перші два елементи записати значення 2.
2. Знайти в масиві m перший (найлівіший) елемент зі значенням 0. Якщо такого елементу немає (змістовно: всі числа в послідовності або підкреслено, або викреслено), алгоритм завершує свою роботу, інакше — продовжує з наступного кроку.
3. Нехай знайдений елемент має індекс i. Записати в нього значення 1.
4. До кінця масиву в усі елементи з індексами, кратними i, тобто в елементи з індексами $k \cdot i < N$ для $k = 2, 3, \ldots$ записати значення 2. Перейти до кроку 2.

Після завершення описаного алгоритму потрібно переглянути масив і знайти в ньому ті елементи, значення яких дорівнює 1. Їх індекси і будуть простими числами, які потрібно було знайти.

Текст програми, яка реалізує описаний алгоритм, наведено нижче. Треба звернути увагу, що за допомогою макроса означено не лише розмір робочого масиву, але й числові коди станів

«чисто», «підкреслено» і «закреслено». Це гарний професіональний підхід, оскільки він дозволяє при першому ж погляді на текст програми зрозуміти її сенс. Порівняємо, наприклад, два оператори: `p[i] = PRIME` та `p[i] = 1`. В першому видно, що елементу масиву присвоюється значення з *предметним змістом* «просте», а в другому — лише «безлике» число 1. Для розуміння задуму програми не так важливо знати, яке саме число використовується, 1 чи 40, важливе лише те, що воно слугує позначенням простого.

```
1  #include <stdio.h>
2
3  #define N  1000
4  #define CLEAR 0 /* англ. "чисте"     */
5  #define PRIME 1 /* англ. "просте"    */
6  #define ERASE 2 /* англ. "видалене" */
7
8  int main() {
9    int p[N], i, j;
10   for( i = 2; i < N; ++i )
11     p[i] = CLEAR;
12   p[0] = ERASE;
13   p[1] = ERASE;
14   i = 0;
15   while( 1 ) { /*  цикл з виходом з середини */
16     while( (i < N) && (p[i] != CLEAR) )
17       ++i; /* пошук першого чистого числа */
18     if( i == N )
19       break; /* якщо масив вичерпано - кінець */
20     p[i] = PRIME; /* знайдене число підкреслити */
21     /* закреслити всі кратні */
22     j = i*2;
23     while( j < N ) {
24       p[j] = ERASE;
25       j += i;
26     }
27   }
28   /* друк результатів:
29       перебирати всі числа */
30   for( i = 0; i < N; ++i )
31     if( p[i] == PRIME ) /* якщо підкреслене */
32       printf( "%d\t", i );
33   printf( "\n" );
34   return 0;
35 }
```

Перший смисловий блок в програмі — початкові присвоювання. Перші два члени послідовності Ератосфена помічаються як вилучені з розгляду, а всі інші — як чисті (нерозглянуті).

Далі йде цикл, у якого замість умови стоїть число 1. Нагадаємо, що умовою циклу в мові С виступає будь-який арифметичний вираз, його значення обчислюється на кожній ітерації і має такий зміст: 0 — припинити цикл, будь-яке інше число — продовжити. Оскільки вираз «1» завжди дає відмінне від нуля значення, цикл мав би продовжуватися нескінченно. Але в тілі циклу є додаткова точка виходу за допомогою оператору **break**.

Змінна `i` відіграє роль поточного числа, кандидата на простоту. В тіло першого циклу вкладено другий цикл, в якому по одному перебираються значення `i` доти, доки не знайдеться чисте (не закреслене) число, або поки не буде досягнуто кінця нашої послідовності чисел. У другому випадку потрібно перервати зовнішній цикл, це і є точка виходу з середини.

Якщо ж змінна `i` ще не досягла прикінцевого значення, число `i` потрібно підкреслити, це значить — присвоїти в `i`-й елемент масиву значення 1 (константа `PRIME`). Далі потрібно закреслювати всі числа послідовності, кратні щойно знайденому простому числу `i`. Для їх перебору слугує змінна `j` та ще один вкладений цикл. Спочатку змінній `j` присвоюється значення `2*i`,

а на кожній ітерації її значення збільшується на `i`, отже, змінна пробігає послідовність значень $2i, 3i, 4i, \ldots$

Нарешті, після закінчення основного циклу виконується цикл, що роздруковує результати. Він перебирає всі числа з послідовності $0, \ldots, N-1$ та друкує лише ті, для яких масив містить помітку «просте».

6.9. Підсумковий огляд

Масиви є одним з найфундаментальніших понять програмування, а їх винайдення ще у першій алгоритмічній мові високого рівня Фортран слід віднести до найвизначніших віх в історії програмування. Відколи в мову програмування залучаються масиви, різко підвищується виразність мови та стрибком розширюється коло задач, які можуть в ній вирішуватися.

Справді, в попередніх главах посібника вивчалися засоби, що дозволяли проводити обчислення над окремими, не пов'язаними між собою змінними. Хоча в програмі могло бути як завгодно багато таких поодиноких змінних, досі залишалося неможливим обробляти сукупність змінних як одне ціле. Таку можливість надають саме масиви.

Далі, в програмах без масивів кожен оператор, так би мовити, заздалегідь «знає», з якою змінною він працюватиме: в операторі записано ім'я змінної, як, наприклад, у присвоюванні `a = b+c`. З появою масивів виникає принципово нова можливість: програма заздалегідь не знає, з якою змінною працює той чи інший оператор, а визначає це під час свого виконання. Наприклад, нехай в програмі є оператор присвоювання `m[k] = b+c`. З самого цього оператору не видно, в яку змінну–елемент масиву він присвоїть значення, в `m[0]`, чи, скажімо, в `m[7]`, це визначається в ході роботи програми, коли та обчислить певне значення індексу у змінній `k`.

Побудова ефективних (швидкодійних та економних за використанням пам'яті) алгоритмів сортування та пошуку була та залишається складною не лише інженерною, але й фундаментально-науковою проблемою. В монументальній праці Д.Кнута «Мистецтво програмування» цілий том (близько тисячі сторінок) має назву «Сортування та пошук» [8], багато алгоритмів пошуку та сортування розглянуто в класичній книзі Н.Вірта [4]. В цьому посібнику розглянуто лише найпростіші алгоритми. Але по-перше, їх повинно бути достатньо для першого знайомства з безмежним колом задач програмування, а по-друге всі більш досконалі алгоритми так чи інакше базуються на описаних тут елементарних ідеях.

Уважний читач міг би помітити, що в цій главі не описано, як передавати масиви до функцій в якості аргументів, та як функції можуть повертати масиви в якості значень. Крім того, розмір масивів, які розглядалися в цій главі, був жорстко зафіксований в тексті програми, тобто програма під час виконання не могла сама визначати потрібний їх розмір масиву. Розгляд цих питань потребує залучення нового та надзвичайно важливого поняття — покажчика.

Розділ 7

Покажчики та динамічна пам'ять

7.1. Основні поняття

Ключові слова: покажчик, адреса, операція розіменування, операція взяття адреси.

Змінні, які розглядалися досі, містили деякі числові значення, цілі або нецілі. В цьому розділі вивчається принципово інший різновид типів даних.

Будь-яка змінна будь-якого типу розміщується у пам'яті. *Покажчик на змінну — це місце її розташування в пам'яті комп'ютера*, іншими словами — її адреса. Якщо, скажімо, змінна типу **int** може зберігати такі значення, як 1, 2, 3, ..., то змінна типу покажчика на ціле може зберігати адресу змінної **int** x, змінної **int** y, змінної **int** z і т.д.

Важлива особливість мови C полягає в тому, що покажчики в ній *типізовані*. Це означає, що якщо змінна, скажімо, p має тип «покажчик на **int**», то значеннями змінної p можуть бути адреси лише змінних типу **int** та не можуть бути адреси змінних типу **double** чи інших. Взагалі, який би тип в мові C не розглянути (позначимо його літерою T), йому відповідає свій тип покажчика — покажчик на T.

Щоб оголосити змінну типу покажчика на деякий тип, потрібно перед іменем змінної поставити зірочку:

```
ім'я_типу *ім'я_змінної_покажчика;
```

В одному оголошенні можна поєднувати звичайні змінні та змінні типу покажчика. Наприклад:

```
int *p;
double *q1, *q2;
int x, *r, s=0, *t;
```

В першому рядку оголошено змінну p типу покажчика на ціле. В другому рядку оголошено одразу дві змінні типу покажчика на дійсні числа: значеннями змінної q1 та змінної q2 можуть бути адреси змінних типу **double**. В третьому рядку оголошено змінну x типу цілого числа, потім змінну r типу покажчика на ціле, далі змінну s знов звичайного цілого типу (з одночасним присвоєнням початкового значення) і, нарешті, змінну t типу покажчика на ціле.

Операція *взяття адреси* & знаходить адресу змінної. Нехай a — ім'я деякої змінної, тоді вираз &a дає адресу змінної a. Наприклад, після оператору присвоєння

```
int *p, x=10;
p = &x;
```

значенням змінної p стане адреса змінної x: вираз в правій частині визначає адресу змінної x, а оператор присвоювання присвоює отриману адресу у змінну p. Тепер, як кажуть, змінна-покажчик p показує на змінну x. Домовимося змінну, на яку вказує покажчик, називати його мішенню. В цьому прикладі мішенню покажчика p є змінна x.

Операція *розіменування покажчика* (її ще називають операцією непрямого звертання) протилежна до операції &: вона дозволяє, знаючи покажчик, знайти ту змінну, на яку він показує. Позначається розіменування зірочкою перед іменем змінної-покажчика. Якщо p — ім'я деякої змінної-покажчика, то вираз *p дає змінну, адресу якої містить p. Вираз *p, де б він не використовувався (щоб уникнути прикрих помилок, пов'язаних з пріоритетами операцій, рекомендується брати його в дужки), має той самий сенс, як ніби замість нього стояла ця змінна-мішень.

Наприклад, у наведеному нижче фрагменті змінна-покажчик `p` вказує на змінну `x`, тому операція `+=` збільшить на 2 саме цю змінну, тепер її значення буде становити 20. В наступному рядку покажчик `p` все ще показує на змінну `x`, тому в змінну `m` буде присвоєно значення 4. Далі в змінну-покажчик `p` присвоюється адреса змінної `m`. Тому наступна операція `+=` працює вже зі змінною `m` (її новим значенням стає число 6). Нарешті, останній рядок має сенс «взяти ту змінну, на яку зараз вказує покажчик `p`, помножити її значення на 3 та надрукувати результат». Очевидно, надруковане буде число 18.

```
1 int x=18, *p, m;
2 p = &x;
3 (*p) += 2;
4 m = (*p) / 5;
5 p = &m;
6 (*p) += 2;
7 printf( "%d\n", 3 * (*p) );
```

З цього прикладу видно, що у програміста з'являється можливість через одну й ту саму змінну-покажчик працювати з різними змінними-мішенями. Так, один і той самий оператор `(*p) += 2` в перший раз обробляв змінну `x`, а в другий — змінну `y`. Завдяки покажчикам програмування набуває особливої гнучкості та виразності.

Отже, якщо `T` — деякий тип мови C, то покажчик на `T` — це такий тип, у якого множина значень охоплює всі можливі адреси змінних типу `T`, а серед операцій є взяття адреси та розіменування (та деякі інші операції, одну з яких буде розглянуто нижче).

Логіка програм з покажчиками суттєво відрізняється від будови програм, де обчислення ведуться лише над «звичайними» змінними. В руках професіонала покажчики — елегантний та потужний інструмент, який дозволяє писати високоефективні і компактні програми, вкладаючи весь розв'язок складної задачі в кілька рядків.

Запитання

1. Що являє собою тип даних «покажчик»?
2. Як зрозуміти, що в мові C покажчики типізовані?
3. Як оголосити змінну типу покажчику на дійсне число?
4. Що таке операція взяття адреси? Операція розіменування покажчика?
5. Як через один покажчик працювати по черзі з кількома змінними? Навести приклад.

7.2. Особливості покажчиків

Ключові слова: «дикий покажчик», порожній покажчик `NULL`, абстрактний покажчик, покажчик на **`void`**.

Запровадження покажчиків до арсеналу програміста не лише дає потужний інструмент програмування та відкриває нові можливості, але й приносить з собою нові різновиди помилок. Характерною помилкою є так званий *«дикий покажчик»*. Це змінна типу покажчика, якій не присвоєно певне значення адреси деякого об'єкту, але з якою програма намагається працювати, наче з покажчиком на дані. Розглянемо програму:

```
int x, y, *p;
y = *p;
*p = 10;
```

Тут оголошується змінна `p` типу покажчика, але значення адреси якоїсь певної змінної їй не присвоюється. Змінна, якій програміст не надав значення власноруч, за допомогою оператору присвоювання, має деяке непередбачуване значення. Наприклад, покажчик `p` може випадково показувати на життєво важливу область даних операційної системи, на змінну іншої програми тощо.

Програма намагається працювати з покажчиком `p`: взяти значення, яке міститься за даною адресою, та присвоїти його у змінну `y`, а потім присвоїти в комірку за цією адресою значенні 10. Зрозуміло, що оскільки покажчик показує невідомо куди, то операції над ним в кращому випадку безглузді, а в гіршому можуть завдати шкоди іншим програмам та самій операційній системі.

Увага! Дикий покажчик може стати причиною серйозних помилок, в тому числі краху програми та операційної системи. Мова С, її транслятор та стандартні засоби не містять захисту від диких покажчиків. Вся відповідальність за роботу з покажчиками лежить на програмісті. Треба слідкувати за тим, щоб перед першим звертанням до покажчика він вказував на якийсь справжній об'єкт даних.

З попереднього повинно бути зрозуміло, що на практиці потрібно якось вирізнити ситуацію, коли покажчик нікуди не показує. Розглянуті вище правильні покажчики показують на ті чи інші об'єкти-мішені, дикий покажчик містить деяку адресу, але за цією адресою не міститься об'єкт. Програма не може відрізнити правильний покажчик від дикого, оскільки в обох випадках покажчик має значення адреси якоїсь комірки, а за адресою машина не може розпізнати, зайнята вона чи вільна.

Для цього слугує спеціальне значення, позначене словом `NULL`. Тепер будь-яка змінна типу покажчика може містити або адресу якогось «справжнього» об'єкта-мішені, або порожнє значення `NULL`, яке означає відсутність мішені у цього покажчика. Для того, щоб використовувати слово `NULL` в програмі, потрібно підключити[1] заголовочний файл `stdlib.h`.

Порожнє значення покажчика зручне тим, що дозволяє програмі гнучко змінювати свою поведінку залежно від того, є чи немає мішень у деякого покажчика:

```
int *p = NULL;
. . .
if( p != NULL ) {
  покажчик не порожній, робота з мішенню;
}
else {
  обробка випадку відсутності мішені;
}
```

Цей принцип буде часто використовуватися в подальшому.

Якщо покажчик `p` порожній, то звертатися до його мішені немає сенсу, бо самої мішені не існує. Спроби такого звертання розцінюються під час виконання програми як помилки.

Наостанок треба розповісти про тип, який посідає особливе місце серед типів покажчиків. Як зазначалося вище, оголошуючи змінну `int *p`, ми обіцяємо транслятору, що в пам'яті там, куди показує цей покажчик, можуть лежати лише цілі числа. Але іноді буває потрібно працювати не з адресою цілого, дійсного і т.д. числа, а з *просто адресою*. Тобто треба, щоб в змінній містилася адреса комірки пам'яті без прив'язки до якогось конкретного типу, до якого мали б належати дані в комірці. Іншими словами, це має бути покажчик на дані заздалегідь не визначеного типу, або абстрактний покажчик.

В мові С абстрактний покажчик позначається як покажчик на тип **void**. Взаємодія між покажчиками конкретних типів та абстрактними покажчиками визначається двома правилами: значення будь-якого конкретного покажчика можна присвоїти абстрактному покажчику; значення абстрактного покажчика можна присвоїти конкретному покажчику лише через операцію перетворення типів.

```
int x=0, *pi1=&x, *pi2; /* покажчик на ціле */
double y=1.1, *pd=&y; /* покажчик на дійсне */
void *p; /* абстрактний покажчик */
p = pi1; /* від конкретного до абстрактного */
pi2 = (int*) p; /* через операцію перетворення */
```

[1] Справа в тому, що константа `NULL` означена як макрос (тобто за допомогою директиви **#define**). Директива з означенням міститься у файлі `stdlib.h`.

Звичайно ж, з операціями перетворення типів треба поводитися обережно, бо вони можуть призвести до помилок. Продовжимо попередній приклад:

```
pd = (double*) p;
*pd = 2.718;
*pi1 = 8;
```

В змінній `p` міститься адреса цілочисельної змінної `x`. Але операція перетворення, фактично, каже транслятору, щоб той розглядав цю адресу так, ніби це адреса змінної дійсного типу. Таким чином, покажчик `pd` вказує на ту комірку пам'яті, де лежить ціле число, але «думає», ніби там міститься дійсне число. Подальші оператори намагаються через два покажчики працювати з однією коміркою пам'яті, розглядаючи її то як ціле, то як дійсне число, що, ймовірніше за все, не те, що мав на увазі програміст.

Запитання

1. Що таке «дикий» покажчик, до яких наслідків він може призвести, як їх уникнути?
2. Що таке порожній покажчик, як він позначається?
3. Що таке абстрактний покажчик? Якими правилами регулюється присвоювання абстрактному покажчику значення конкретного покажчика та навпаки?
4. В чому небезпека необережного використання абстрактних покажчиків?

7.3. Покажчики як аргументи та значення функцій

Ключові слова: нелокальне присвоювання.

Багаті можливості перед програмістом відкриває використання покажчиків в ролі аргументів функцій. Розглянемо програму:

```
1  #include <stdio.h>
2
3  void swap( int *p, int *q );
4  int main() {
5  int x=12, y=43;
6  swap( &x, &y );
7  printf( "x=%d,␣y=%d\n", x, y );
8  return 0;
9  }
10
11 void swap( int *p, int *q ) {
12 int t;
13 t = *p;
14 *p = *q;
15 *q = t;
16 }
```

Функція `swap`[1] має два аргументи типу покажчиків на цілочисельні змінні. У функції `main` оголошено локальні змінні `x` та `y`. З функції `main` викликається функція `swap`, причому в якості аргументв їй передаються адреси змінних (звернути увагу на знак `&` перед обома змінними).

Оператори в тілі функції `swap` фактично означають: взяти значення з комірок пам'яті, на які вказують покажчики `p` та `q`, та обміняти їх місцями. Викликана функція `swap` отримує значення аргументів: аргумент `p` містить адресу змінної `x` з функції `main`, а аргумент `q` — адресу змінної `y`. Отже, після повернення з виклику назад до функції `main` значення її змінних виявляться переставленими: `x` матиме значення 43, а `y` — значення 12.

Можна згадати, що згідно осовних засад мов C та й взагалі парадигми структурно-модульного програмування, стороння функція не може за власним бажанням втручатися

[1] swap — англ. обміняти місцями

у локальні змінні іншої функції. Але в даному випадку одна функція, коли викликає іншу функцію, за власним бажанням передає їй адреси своїх локальних змінних, а отже і право доступу до цих змінних. Іншими словами, передача адрес змінних до іншої функції в якості аргументів дозволяє робити *нелокальні присвоювання.*

Передача адрес локальних змінних через аргументи до іншої функції належить до тих прийомів програмування, які слід використовувати з обережністю. Недбале поводження з таким засобом може призвести до серйозних помилок в логічній будові програми, які дуже важко знайти та виправити, або зробити будову програми і логіку її роботи важкою для розуміння. Справді, коли функція може отримувати доступ до локальних змінних іншої функції, це, строго кажучи, порушує фундаментальний принцип структурно-модульного програмування «розділяй та володарюй» і створює додаткові сильні зв'язки між функціями. Разом з тим, там, де це дійсно виправдано задачею, цей прийом може сприяти дуже елегантним рішенням. В підсумку, користуватися аргументами-покажчиками можна, але обмежено.

Тепер можна пояснити, навіщо при аргументах функції `scanf` (крім форматного рядка) стоїть знак `&`. В якості аргументів до функції передаються *адреси* місцезнаходження в пам'яті тих змінних, в які функція `scanf` має розмістити введені дані.

Функція також має право повертати значення типу покажчика, наприклад нижче показано функцію, яка приймає на вхід два покажчика на цілочисельні змінні та повертає той з покажчиків, який вказує на більше число.

```
int *max_ptr( int *a, int *b ) {
if( *a > *b ) return a;
return b;
}
```

Покажемо приклад використання цієї функції. Нехай десь у програмі обчислюються значення змінних `p` та `q`, і ту з них, у якої значення виявиться більшим, треба обнулити.

```
int p, q, *p;
/* обчислення p i q */
p = max_ptr( &a, &b );
*p = 0;
```

В змінну `p` потрапляє адреса однієї з двох змінних, `p` чи `q`. В останньому рядку тій змінній, на яку вказує покажчик `p`, присвоюється значення 0. Особливо цікаво, що два останні рядки можна замінити на один, якщо операцію розіменування застосувати прямо до результату функції, без допоміжної змінної `p`:

```
*ptr( &a, &b ) = 0;
```

Справді, оскільки значенням функції є покажчик, то операція розіменування, застосована до виклику функції, дає змінну, яка має право стояти в лівій частині операції присвоювання.

Тут знов, як це нерідко траплялося в попередніх розділах, надзвичайна потужність та гнучкість засобів мови при необдуманому використанні може стати джерелом серйозних помилок. Розглянемо приклад:

```
1  int *func() {
2  int x = 0;
3  return &x;
4  }
5
6  int main() {
7  int *p;
8  p = func();
9  *p = 0;
10 return 0;
11 }
```

З точки зору граматики мови текст функції `func` абсолютно правильний, але функція є міною уповільненої дії. Змінна `x` — це локальна змінна даної функції. Треба пригадати, що

локальна змінна створюється при вході в функцію та знищується при поверненні з неї. Функція `func` повертає до функції `main` адресу своєї локальної змінної, але в момент повернення змінна перестає існувати. В результаті функція `main` отримає адресу того місця в пам'яті, де щойно *була* змінна, і значення 0 буде присвоєно вже у незайняту область пам'яті. Більше того, якщо з покажчиком `p` намагатися робити якісь дії далі, то можливо, що ця область пам'яті буде вже повторно виділена під локальні змінні іншої функції, тоді функція `main` зіпсує чужі змінні.

Таким чином, функція ніколи не повинна повертати адреси своїх локальних змінних.

Запитання

1. Що таке нелокальне присвоювання та як його зробити за допомогою аргументів типу покажчика?
2. В чому використання аргументів-покажчиків суперечить засадам структурно-модульного стилю програмування?
3. Написати функцію, яка отримує аргумент типу покажчика на цілочисельну змінну та збільшує значення цієї змінної на 1.
4. Написати функцію *впорядкування пари*, яка має два аргументи, `u` та `v` типу покажчиків на дійсні змінні. Функція бере значення, на які вказують покажчики-аргументи, та робить так, що в результаті виклику функції менше з цих чисел опиняється в змінній, на яку вказує `u`, а більше — у змінній, на яку вказує `v`.
5. Обґрунтувати, чому функціям не слід повертати покажчики на свої локальні змінні.

7.4. Покажчики та масиви

Найхарактерніше застосування покажчиків в мові C пов'язане з масивами. Фактично, ім'я масиву і покажчик на перший елемент масиву — це, з точки зору транслятора, одне й те саме.

Важлива особливість реалізації масивів полягає в тому, що елементи масиву розташовані в пам'яті підряд, один за одним. Справді, було б дуже не логічно зберігати, скажімо, перший елемент масиву в комірці номер 1018, другий — в комірці 994, третій — в комірці 2204 тощо. Нехай відома адреса першого елементу масиву, `&m[0]`. Тоді для того, щоб знайти елементи з індексом 1, 2, k, потрібно до цієї адреси додати зміщення — ту кількість комірок, яку займають відповідно, один, два або k елементів.

Найважливіше правило полягає в тому, що в мові C покажчик на початок масиву можна використовувати всюди, де може стояти ім'я масиву, і до покажчика можна застосувати операцію індексування. Наприклад:

```
int m[10], *p, k=2;
p = & m[0];
p[k] = 8;
```

В першому рядку оголошено масив цілих чисел `m`, змінну-покажчик `p` та цілочисельну змінну `k` з початковим значенням. Далі у змінну-покажчик `p` присвоюється адреса першого елементу масиву. Тепер у третьому рядку операція індексування «знайти елемент під номером `k`» застосовується до покажчика. Оскільки він вказує на початок масиву, то значення буде присвоєно у третій (той, що має індекс 2) елемент масиву `m`.

Ім'я масиву саме по собі є покажчиком-константою. Тому в наведеному прикладі другий рядок можна було б записати простіше: `p = m`. Єдина відмінність між змінною-покажчиком на початок масиву `p` та іменем самого масиву `m` полягає в тому, що `p` є змінною, їй можна присвоювати значення (ту чи іншу адресу), а `m` є константою і тому не може стояти в лівій частині присвоювання.

Завдяки принципу «масив — це покажчик» з'являється надзвичайно важлива для практичних застосувань можливість: передавати до функції в якості аргументів покажчик на початок масиву та кількість елементів в ньому. Цей спосіб гнучкий та універсальний, оскільки розмір масиву не фіксується жорстко заздалегідь, а передається через аргумент, отже одну й ту саму функцію можна застосовувати до різних масивів.

Наприклад, розглянемо програму, в якій функція обчислення середнього арифметичного елементів масиву викликається двічі для масивів різної довжини.

```
1  /* прототип функції */
2  double average( double *p, int n );
3  #define M 10
4  #define N 20
5  int main() {
6  double a[ M ], b[ N ], u, v;
7  . . . /* введення елементів */
8  /* два виклики для різних масивів */
9  u = average( a, M );
10 v = average( b, N );
11 . . .
12 return 0;
13 }
14
15 /* реалізація функції */
16 double average( double *p, int n ) {
17 double s = 0;
18 int i;
19 for( i = 0; i < N; ++i )
20   s += p[i];
21 return s/n;
22 }
```

Тут функція `average` для обчислення середнього арифметичного має два аргументи: перший — покажчик на початок того масиву, з елементів якого треба знайти середнє, а другий — кількість елементів в цьому масиві. Тіло функції не потребує особливих коментарів: в циклі елемент за елементом обчислюється сума, функція повертає частку від ділення цієї суми на кількість елементів. Звернімо увагу на те, що до покажчика `p` застосовується операція індексування `[]`: покажчик на початок масиву та власне масив поводять себе повністю однаково.

У тілі функції `main` оголошено два масиви різної довжини. Спочатку функція `average` викликається для масиву `a`, в аргумент `p` передається покажчик на початок цього масиву (нагадуємо, що ім'я масиву саме по собі є константою-покажчиком), а в аргумент `n` — довжина масиву, число `M`, що дорівнює 10. Потім ця ж функція викликається для другого масиву, в аргументи передаються його початок та довжина.

Наведемо ще один приклад. Нехай потрібно створити функцію додавання векторів в n-вимірному просторі для довільного n. Програмною моделлю такого вектору є, звичайно ж, масив дійсних чисел довжиною n елементів. Додавання векторів має сенс лише тоді, коли вони мають однакову розмірність, таку ж розмірність матиме і їх сума. За означенням компоненти суми визначаються за формулою $s_i = a_i + b_i$ для всіх $i = 1, 2, \ldots, n$.

Отже, функція повинна отримувати такі аргументи: покажчик на початок масиву, в який треба записати суму, два покажчики на початки масивів, що моделюють вектори-доданки, та ціле число, кількість елементів. Не забудемо, що в математиці компоненти вектору прийнято нумерувати від 1 до n, а в мові C елементи масиву нумеруються від 0 до $n - 1$. Отримуємо такий текст:

```
void add_vect( double *s, double *a,
  double *b, int n ) {
int i;
for( i = 0; i < n; ++i )
  s[i] = a[i] + b[i];
```

```
}
```

Звернімо увагу на те, що ця функція не повертає значення (тип значення вказано як **void**), оскільки результати своїх обчислень вона розміщує в пам'яті за адресами починаючи з `s`.

Запитання

1. Чому для того, щоб обробляти будь-який елемент масиву, достатньо знати лише адресу першого елемента?
2. Яка операція, характерна для масиву, може застосовуватися до покажчика?
3. Як масив передати до функції в якості аргументу?
4. Розглянуті раніше алгоритми лінійного пошуку у невпорядкованому масиві, двійкового пошуку у впорядкованому масиві, алгоритм сортування реалізувати у вигляді функцій (нехай ці функції працюють з масивами цілих чисел).

7.5. Динамічна пам'ять

Ключові слова: виділення пам'яті, звільнення пам'яті, `alloc.h`, **sizeof**, `malloc`, `free`.

Програми, які розглядалися раніше, працювали зі заздалегідь відомими та жорстко зафіксованими в тексті програми розмірами масивів. Кількість елементів в оголошеннях на кшталт **int** `a[20]` відома ще на етапі трансляції програми і не може бути змінена динамічно, в процесі її виконання. Крім того, виділення та звільнення пам'яті здійснювалося автоматично — програміст не мав контролю за цим процесом.

Засоби динамічного управління пам'яттю дозволяють різко розширити можливості програмування та принципово збільшують потужність мови. Програма отримує можливість під час виконання самостійно визначати потрібні розміри пам'яті, «на льоту» створювати масиви та знищувати їх за власним бажанням.

Загальну логіку роботи програми з пам'яттю добре пояснює така метафора. Прикладна програма, маючи потребу в додатковій області пам'яті, звертається за послугою до внутрішніх, прихованих від очей користувача, механізмів підтримки, які відповідають за управління всією доступною пам'яттю. Формою такого звернення за послугою є виклик спеціальної функції.

Перейдемо до конкретного розгляду техніки роботи з динамічною пам'яттю. Нижче наведено текст програми, яка вводить з клавіатури ціле число `n`, динамічно створює масив дійсних чисел розміром в `n` елементів та вводить елементи масиву (подальшу обробку масиву в цій програмі не розглядаємо). Рядки, на які треба звернути увагу, відмічені коментарями.

```
1  #include <stdio.h>
2  /* для роботи з динам. пам'яттю */
3  #include <alloc.h>
4  int main() {
5  double *p; /* динамічний масив - покажчик! */
6  int n; /* кількість елементів - змінна! */
7  int i;
8  printf( "Кількість␣елементів?␣" );
9  scanf( "%d", &n );
10 /* !!! створення динамічного масиву !!! */
11 p = (double*) malloc( n * sizeof(double) );
12 /* перевірити, чи масив створився! */
13 if( p == NULL ) {
14   printf( "Нестача␣пам'яті\n" );
15   return -1; /* програма завершується */
16 }
17 /* якщо виконання дійшло до цієї точки,
18 значить масив створився і змінна p
```

```
19 показує на його початок */
20 printf( "Введіть␣елементи␣масиву\n" );
21 for( i = 0; i < n; ++i ) {
22   printf( "%d-й", i );
23   scanf( "%lf", &p[i] );
24 }
25 . . .
26 /* подальша обробка масиву */
27 . . .
28 free( p ); /* знищити масив */
29 return 0;
30 }
```

Пояснімо основні деталі цієї програми. Динамічне управління пам'яттю в мові C здійснюється за допомогою стандартних функцій, оголошення яких містяться в заголовочному файлі `alloc.h`.

Робота з динамічними масивами здійснюється через змінну типу покажчика, в даній програмі для цього слугує змінна `p`. Сенс полягає в тому, що після створення масиву ця змінна буде показувати на його початок, тобто міститиме адресу його першого елементу. Тоді через змінну-покажчик з динамічним масивом можна буде працювати так само, як пояснювалося в попередніх параграфах.

Звернімо увагу, що розмір динамічного масиву задано вже не константою, а змінною `n`. Програма не може зазделегідь знати, яке значення отримає ця змінна. В наведеному прикладі розмір масиву вводить з клавіатури користувач. В інших задачах програма може також і сама обчислювати розміри потрібних їй масивів за деякими формулами.

Для виділення шматка динамічної пам'яті використовується функція `malloc`. Вона приймає один аргумент цілого типу — розмір в байтах області пам'яті, яку потрібно виділити. Обчислити розмір масиву можна, звичайно ж, помноживши кількість елементів масиву на розмір в байтах одного елементу. Для того, щоб у програмі дізнатися кількість байтів, яку займають ті чи інші дані, слугує операція **`sizeof`**. Вираз вигляду **`sizeof`**`(тип)`, де `тип` — ім'я будь-якого типу даних, компілятор перетворює на ціле число — розмір одного об'єкту цього типу.

Функція `malloc` виділяє у вільній пам'яті область замовленого розміру і відмічає її як зайняту. Щоб передати цю область пам'яті у володіння програмі, функція `malloc` повертає адресу початку області, саме ця адреса присвоюється в змінну-покажчик `p`.

Область пам'яті, виділену за допомогою функції `malloc`, програма може використовувати, в принципі, будь-яким чином. Найчастіше програма обробляє її як масив елементів, хоча можуть бути й інші способи.

Оскільки функція `malloc` не знає заздалегідь, дані якого типу будуть розміщені у динамічно виділеній області пам'яті, то значення, яке вона повертає, має тип абстрактного покажчика **`void`**`*`. Тому для того, щоб використати його, як в даному прикладі, в ролі масиву дійсних чисел, його треба перетворити до відповідного типу, в даному випадку до типу **`double`**`*`. Для цього слугує операція перетворення типу, що в тексті програми записана перед іменем функції `malloc`.

Функція `free` призначена для звільнення області динамічної пам'яті, яку раніше було виділено. Її єдиний аргумент — це покажчик на область, яку треба звільнити.

Запитання

1. Що таке динамічний розподіл пам'яті, чим він відрізняється від того, який застосовувався в попередніх розділах?
2. Який заголовочний файл треба підключити, щоб зробити доступними засоби динамічного управління пам'яттю?
3. Для чого слугує операція **`sizeof`**?

4. За допомогою якої функції можна динамічно виділити область пам'яті заданого розміру? Як визначити розмір пам'яті для масиву з n дійсних чисел?
5. Що собою являє тип **void***?
6. Яка функція звільняє динамічно виділену пам'ять?
7. Програми лінійного, двійкового пошуку та сортування перетворити таким чином, щоб число N вводилося з клавіатури, а масив довжини N створювався динамічно.
8. В попередніх розділах було розібрано програму пошуку методом решета Ератосфена всіх простих чисел, що не перебільшують N, в якій число N жорстко зафіксоване. Перетворити її на функцію, яка сприймає довільний цілий додатній аргумент N.

7.6. Особливості динамічно створених даних

Динамічно створені об'єкти даних (в тому числі масиви) помітно відрізняються від змінних, пам'ять під які виділяється автоматично. Динамічні об'єкти не є локальними для якоїсь функції (локальною може бути змінна, яка містить адресу такого об'єкта). Це значить, що динамічний об'єкт може створюватися в одній функції, обробляти його може друга функція (якщо перша повідомить їй адресу об'єкта), а знищувати може третя. Іншими словами, доступ до динамічного об'єкта має будь-яка функція програми, якій відома його адреса. Час життя динамічно створеного об'єкту — це час від його створення функцією `malloc` до знищення функцією `free`, а вони можуть викликатися як в одній функції програми, так і в різних (як вирішить програміст). Проілюструємо сказане прикладом.

```
1  #include <alloc.h>
2
3  double *alloc_double_array( int n );
4  void kill_double_array( double *p );
5  double *make_fibonacci_table(double *p, int n);
6
7  int main() {
8  double *fibon;
9  int m, i;
10 printf( "кількість␣наближень?␣");
11 scanf( "%d", &m );
12 fibon = make_double_array( m );
13 make_fibonacci_table( p, m );
14 for( i = 1; i < m; ++i )
15   printf( "%lf", fibon[i] / fibon[i-1] );
16 kill_double_array( fibon );
17 return 0;
18 }
19
20 double *alloc_double_array( int n ) {
21 return (double*) malloc( n * sizeof(double) );
22 }
23
24 void kill_double_array( double *p ) {
25 free( p );
26 }
27
28 double *make_fibonacci_table(double *p, int n) {
29 int i;
30 p[0] = 1;
31 p[1] = 1;
32 for( i = 2; i < n; ++i )
33   p[i] = p[i-1] + p[i-2];
```

```
34 }
```

Функція alloc_double_array виділяє пам'ять для масиву з n дійсних чисел (де число n передається як аргумент) та повертає покажчик на нього. Власноруч створити на основі виклику функції malloc свою власну функцію для виділення пам'яті під масив — корисний прийом, який варто використовувати в своїх програмах. Справа в тому, що функція malloc надто громіздка — щоразу при її виклику треба обчислити розмір масиву, потім перетворити результат на потрібний тип конкретного покажчика. Якщо в програмі треба багато разів створювати динамічні масиви, використання функції malloc стає обтяжливим. Натомість, функція alloc_double_array має аргумент зі зрозумілим сенсом — кількість елементів масиву, а значення, яке повертає функція, вже приведене до потрібного типу.

Функція make_fibonacci_table приймає два аргументи: покажчик на масив дійсних чисел та кількість елементів в ньому. Функція заповнює масив числами Фібоначчі.

Спочатку викликається функція alloc_double_array, яка ствфює масив та повертає покажчик на нього. Адреса створеного масиву потрапляє в змінну fibon, локальну змінну функції main. Потім функція main повідомляє цю адресу функції make_fibonacci_table, яка заповнює масив значеннями. Після деякої обробки масиву функція main викликає функцію kill_double_array та передає покажчик на масив їй. Ця функція і знищує масив.

Отже, масив, створений динамічно в одній функції, обробляється в інших функціях. Для цього перша функція має просто повернути покажчик на створений нею масив.

Робота з динамічною пам'яттю потребує від програміста ретельності, уваги і того, що можна назвати професійною культурою. Одна типова помилка — це спроба звернутися за покажчиком до області пам'яті, яку ще не виділено (змінну-покажчик оголошено, але функцію malloc ще не викликано, тому покажчик вказує «в порожнечу», тобто на випадкове місце в пам'яті) — це звичайно призводить до краху програми. Друга типова помилка — спроба працювати з вже звільненою областю, результатом чого також стає крах програми. Нарешті, якщо забути про звільнення динамічної пам'яті, ця область так і залишиться зайнятою, хоча програма вже не збирається її використовувати. До краху це не призводить, але, якщо програма робить так часто, може призвести до великих зайвих затрат пам'яті, що знизить продуктивність роботи системи. Тому потрібно чітко уявляти логічну структуру програми: де і коли динамічна пам'ять виділяється та знищується.

7.7. Покажчики вищих рівнів

Як було сказано вище, який би не взяти тип T, в мові C має сенс також і відповідний тип покажчика T*. Але ж цей принцип можна розповсюдити і на самі покажчики: оскільки є тип покажчика T*, то має сенс і тип покажчика на покажчик T**, тип покажчика на покажчик на покажчик T*** і т.д. Отже, в мові C кожному типу T відповідає в принципі нескінченна башта типів покажчиків.

Розглянемо властивості такої башти на прикладі типу **int**. Нехай оголошено змінні з початковими значеннями:

```
int x0 = 1;
int *x1 = &x0;
int **x2 = &x1;
int ***x3 = &x2;
```

Змінна x0 містить власне дані — ціле число 1. Змінна x1 містить адресу в пам'яті тієї комірки, де розташувалася змінна x0. Але ж x1 сама є змінною і займає певне місце в пам'яті. Змінна x2 містить адресу, по якій розташована змінна x1. Таким же чином, змінна x3 містить адресу, за якою міститься в пам'яті змінна x2.

Робота з покажчиками вищих рівнів в принципі нічим не відрізняється від роботи зі звичайними покажчиками (як їх ще називають, покажчиками першого рівня). Якщо змінна p має тип покажчика n-го рівня, то вираз &p має тип покажчика $(n+1)$-го рівня і його значенням

є адреса змінної `p`. Значенням виразу `*p` є значення змінної, адреса якої міститься в змінній `p`, причому це значення належить до типу покажчика $(n-1)$-го рівня.

Повернувшися до наведеного вище прикладу, розглянемо вираз `***x3`. Перетворимо його, показавши дужками порядок застосування операцій розіменування, вийде вираз `*(*(*x3))`. З цієї форми запису стає очевидним його сенс:

- Взяти змінну `x3` і розіменувати її, тобто знайти ту змінну, на яку вона вказує;
- Взяти значення змінної, отриманої в попередньому пункті. Воно є покажчиком типу **`int**`**. Розіменувати цей покажчик, тобто знайти змінну, на яку він вказує;
- Взяти значення змінної типу **`int*`**, знайденої в попередньому пункті, та розіменувати його. Отримаємо змінну цілого типу.

Отже, для того, щоб отримати ту змінну, на яку в остаточному рахунку вказує покажчик n-го рівня, потрібно застосувати операцію розіменування n разів.

Запитання

1. Що таке покажчик вищого рівня?
2. Як позначається змінна типу «покажчик на покажчик на покажчик на дійсне»?
3. Як присвоїти значення змінній з попереднього питання?
4. Як отримати доступ до дійсного числа через покажчик з питання 2?

7.8. Багатовимірні масиви

Масиви, що розглядалися вище, являли собою лінійну послідовність елементів, ніби записаних підряд на довгій стрічці. Для звертання до будь-якого елементу такого масиву достатньо вказати одне число — номер елементу від початку масиву. Тому такі масиви називають одновимірними. В багатьох практичних задачах, однак, виникає потреба обробляти набори даних з більш складною організацією, наприклад прямокутні таблиці, в яких для звертання до кожного елементу потрібно вказати два числа: номер рядка та номер елементу в цьому рядку (або, що те ж саме, номер стовпчика). Відповідно, такі масиви називають двовимірними. Тривимірниий масив можна уявити у вигляді стосу аркушів з прямокутними таблицями однакового розміру.

В мові C є стандартна підтримка багатовимірних масивів з фіксованими (константними) розмірами та з автоматичним виділенням пам'яті, а також можливість власноруч моделювати багатовимірні масиви за допомогою покажчиків вищих рівнів та динамічного управління пам'яттю. Першому способу сильно бракує гнучкості, тому тут розглянемо лише другий.

Основна ідея обробки динамічних багатовимірних масивів полягає в тому, що n-вимірний масив можна зобразити як одновимірний масив покажчиків на $(n-1)$-вимірні масиви. Наприклад, двовимірний масив (прямокутну матрицю) можна уявити як одновимірний масив покажчиків на одновимірні масиви чисел — рядки матриці. Подібним чином, тривимірний масив моделюється одновимірним масивом покажчиків на двовимірні масиви.

Одновимірний масив дійсних чисел в мові C обробляється через покажчик першого рівня **`double*`** — адресу першого елементу. Тоді двовимірний масив, що є одновимірним масивом покажчиків на одновимірні масиви чисел, повинен оброблятися через покажчик другого рівня **`double**`**. Та й взагалі, n-вимірний масив дійсних чисел моделюється покажчиком n-го рівня **`double`** $\underbrace{\texttt{*...*}}_{n\ \text{разів}}$.

Роботу з багатовимірними масивами через покажчики покажемо на прикладі. Розглянемо програму, яка вводить з клавіатури два числа, m та n, і динамічно створює матрицю з m рядків та n стовпчиків. Потім програма заповнює матрицю значеннями: (i,j)-му елементу присвоюється значення $100 \cdot i + j$. Далі програма здійснює якусь обробку матриці (в тексті прикладу пропущено) та, нарешті, знищує матрицю, звільняючи виділену для неї пам'ять.

```
#include<alloc.h>
#include<stdio.h>
int main() {
int m, n; /* розміри */
int i, j;
double **p; /* покажчик на матрицю */
printf( "Введіть розміри матриці " );
scanf( "%d %d", &m, &n );
/* виділяється під масив покажчиків на рядки */
p = (double**) malloc( m * sizeof(double*) );
/* для кожного рядка виділяється пам'ять */
for( i = 0; i < m; ++i )
  p[i] = (double*) malloc(n * sizeof(double));
/* робота з елементами матриці */
for( i = 0; i < m; ++i )
  for( j = 0; j < n; ++j )
    p[i][j] = 100 * i + j;
. . .
/* знищення матриці */
/* спочатку знищуються всі рядки */
for( i = 0; i < m; ++i )
  free( p[i] );
/* знищується масив покажчиків на рядки */
free( p );
return 0;
}
```

Як було вже сказано вище, матриця зберігається в пам'яті по рядках, кожен рядок в окремому одновимірному масиві, а в ще одному масиві зберігаються покажчики на ці рядки. Створювати треба спочатку масив покажчиків на одновимірні масиви чисел, що робиться в рядку 10, а потім самі ці масиви, що робиться в тілі циклу в рядку 13. Треба звернути увагу на те, що в рядку 10 створюється масив покажчиків на числа, тобто типом елементу масиву є `double*`, що і стоїть під операцією `sizeof`. Масив покажчиків має тип покажчика на покажчик, при функції `malloc` стоїть відповідна операція перетворення типів.

Після того, як матрицю створено, з нею можна працювати простим та очевидним чином. Для звертання до (i, j)-го елементу використовується конструкція `p[i][j]`. Розглянемо детальніше її сенс. З двох записаних підряд операцій індексування (`[i]` та `[j]`) першою виконається та, що стоїть безпосередньо при покажчику. Іншими словами, спочатку буде обчислено вираз `p[i]`, а потім до його результату буде застосовано операцію `[j]`.

Значенням виразу `p[i]` є i-й елемент масиву `p` — одновимірного масиву покажчиків на рядки, тобто покажчик на i-й рядок матриці. З цього рядку операція `[j]` вибирає j-й елемент.

Запитання

Створити в програмі матрицю (двовимірний масив) цілих чисел розміром m рядків на n стовпчиків (числа m та n вводить користувач). Ввести значення елементів матриці з клавіатури. Роздрукувати елементи так, щоб на екрані вони виглядали, як матриця (кожен рядок матриці друкується на окремий рядок екрану, кожен елемент в рядку займає однакову кількість знакомісць). Знайти в матриці найбільший елемент, надрукувати номер його рядка та стовпчика. Підрахувати та надрукувати кількість нульових елементів.

7.9. Підсумковий огляд

Покажчик являє собою адресу в пам'яті (фактично, номер комірки), де зберігається певне значення. Наприклад, змінна типу покажчика на ціле (`int *p`), на відміну від цілої змінної (`int x`), містить не саме ціле число, а місце, адресу, де зберігається ціле. Найважливішими

операціями над покажчиками є: взяття покажчика (знаючи змінну, знайти адресу, де вона розташована) та розіменування (знаючи покажчик, знайти змінну, на яку він вказує).

Покажчики — один з найхарактерніших засобів мови C. Можна навіть сказати, що справжнє (не іграшкове) програмування мовою C це завжди робота з покажчиками. Покажчики мають багато різноманітних застосувань, деякі (але далеко не всі) з яких тут розглянуто.

Покажчик дозволяє (там, де це дійсно потрібно за логікою задачі) обійти правила розмежування областей видимості локальних змінних. Якщо функція `f` передає до функції `g` значення своєї локальної змінної `x`, то що б не робила з ним функція `g`, значення самої змінної від цього не зміниться, бо до функції `g` передається не сама змінна, а тимчасова копія її значення. Але якщо функція `f` передає до функції `g` покажчик на свою змінну `x`, то тим самим вона передає і право присвоювати значення цій змінній: знаючи адресу в пам'яті, функція `g` може в комірку за цією адресою покласти нове значення, а після повернення функція `f`, звичайно ж, «побачить» це значення у змінній `x`.

Існує тісний зв'язок між покажчиками та масивами: транслятор, фактично, не розрізняє поняття «масив» та «покажчик на перший елемент масиву». Завдяки цьому стає можливим передавати масив як аргумент до функції та повертати масив як значення функції.

За допомогою покажчиків стає можливим динамічно (тобто під час виконання програми) створювати та знищувати області даних (частіше за все — масиви), розмір яких був заздалегідь невідомий. Створення таких даних означає виділення пам'яті для них, знищення даних — звільнення пам'яті. Для роботи з динамічно виділеними даними також використовуються покажчики. Так, функція `malloc` видяляє динамічну область пам'яті заданого розміру та повертає покажчик на її початок (адресу комірки, з якої ця область починається).

За допомогою покажчиків на покажчики з'являється можливість підтримувати багатовимірні масиви. Наприклад, двовимірний масив (матрицю) можна змоделювати як одновимірний масив покажчиків на одновимірні масиви — рядки цієї матриці.

Розділ 8

Обробка текстових рядків

Засоби мови C, викладені в попередніх розділах, дозволяють вирішувати в принципі будь-які задачі чисто обчислювального типу, тобто задачі, які зводяться до маніпуляцій з числами та сукупностями чисел за формулами як завгодно високої складності.

Разом з тим, обчислювальні задачі, які становили левову частку програмування на початку комп'ютерної ери, вже давно поступаються місцем задачам зовсім іншого класу — задачам обробки текстів. Так, компілятор мови C є програмою, що обробляє тексти програм. Прикладні програми кінцевого користувача, такі як текстові редактори, також обробляють тексти. Нарешті, перспективна область, що зараз активно розвивається, web-програмування, ґрунтується на обробці текстів мовою HTML — вихідних кодів web-сторінок.

За різними оцінками, задачі текстової обробки займають до 80% задач сучасного програмування.

Для текстової обробки створено спеціальні мови (awk, Perl, вітчизняна розробка — мова Refal, яка зараз переживає своє третє народження), в які вбудовано потужні спеціалізовані операції над текстами. Мова C належить до різновиду універсальних мов. Це означає, що потужних засобів, спеціально призначених для складної обробки текстів, в ній немає, але стандартна бібліотека містить деякі елементарні засоби, яких вистачає для широкого кола не надто складних задач цієї області.

8.1. Символьний тип

Ключові слова: код символа, кодова таблиця.

Перш ніж займатися обробкою текстів, потрібно розібрати тип даних, що призначений для обробки однієї окремої літери (символу). Значення цього типу відіграють роль цеглинок, з яких будуються тексти.

Обчислювальна машина за самою своєю будовою пристосована для обробки чисел. Тому символи в пам'яті зображуються числовими кодами відповідно до *кодової таблиці*. Тим самим обробка символів зводиться до обробки чисел.

Для обробки кодів символів призначений спеціальний тип **char**. Він відрізняється від типу **int** тим, що займає в пам'яті лише один байт, відповідно його діапазон значень становить від -128 до $+127$. Набір операцій, допустимих для даного типу, включає ті ж самі, що і для типу **int**, арифметичні операції додавання, віднімання, множення, ділення, залишку, а також операції порівняння та логічні операції. Іншими словами, тип **char** можна використовувати не лише за прямим призначенням, для обробки символів, але і для обробки просто чисел, якщо програміст може заздалегідь гарантувати, що ці числа входять в діапазон $-128\ldots+127$.

Константи типу **char** можна задавати, по-перше, як і константи типу **int**, у формі десяткових позначень, наприклад

```
char c = 10;
```

Крім того, константу символьного типу можна записати, взявши в одинарні лапки будь-який символ. Наприклад, у фрагменті

```
char h = 'Q';
```

змінній `h` присвоюється значення — число, що є кодом великої літери Q у кодовій таблиці.

Увага! Треба застерегти від типової помилки. Константи `1` та `'1'` це зовсім різні значення. Перше — це число 1, а друге — код друкованого знаку «1», що має зовсім інше значення (в широко розповсюдженій кодовій таблиці ASCII це 49).

Корисно буде розібрати приклад програми, яка друкує на екран код заданого символу.

Табл. 8.1. Функції класифікації символів

Функція	Сенс	Приклади
isalpha	літери абетки	a, A, z, Z
isdigit	цифри	0, 9
isalnum	літера або цифра	a, Z, 4
islower	мала літера	a, r, z
isupper	велика літера	Z, R, Z

```
#include <stdio.h>

int main() {
char c = '1';
int n = c;
printf( "%d\n", n );
return 0;
}
```

Тут оголошено змінну `c` символьного типу, їй присвоєно значення коду символу «1», потім це значення копіюється в змінну `n` цілого типу. Тепер змінні `c` та `n` мають однакові числові значення, а відрізняються лише тим, що перша має тип **char** і займає 1 байт, а друга має тип **int** та займає 2 байти. Отже, сенс цього присвоювання та допоміжної змінної в тому, щоб перетворити значення з символьного типу на цілий. Далі отримане ціле значення друкується звичайним чином. Якщо скористатися операцією перетворення типів, то цей приклад можна скоротити, звільнившись від допоміжної змінної `n`:

```
printf( "%d\n", (int) c );
```

8.2. Функції класифікації. Порядок символів

В задачах обробки текстової інформації дуже часто доводиться розпізнавати, чи належить даний символ тому чи іншому різновиду, наприклад, чи є він літерою алфавіту, великою чи малою літерою, цифрою, знаком пунктуації тощо.

Для таких типових задач стандартна бібліотека містить спеціальне сімейство функцій. Їх імена починаються з англійського слова `is`, за яким слідує назва різновиду символів. Наприклад, `isdigit` (дослівно: чи є цифрою) або `isspace` (чи є пробілом).

Всі ці функції мають один аргумент цілого типу — код символа `c` та повертають ціле число: 1, якщо символ `c` належить даному різновиду символів, та 0 в іншому випадку. Наприклад, функція `islower` повертає значення 1 тоді і тільки тоді, коли її аргумент є кодом деякої малої літери.

Неповний перелік функцій класифікації символів наведено в табл. 8.1.

На превеликий жаль, правильна робота класифікаційних функцій типу `islower` з неанглійськими літерами, зокрема з літерами кирилічної абетки, гарантується не в усіх реалізаціях мови C.

Кодові таблиці (способи нумерації символів) побудовано таким чином, що англійські літери та цифри розташовані у своєму природному порядку. Якщо з двох літер перша стоїть в абетці раніше за другу, то і її числовий код менше за код другої літери. Якщо друга літера стоїть в абетці одразу ж після першої (як, наприклад, літера q після літери p), то і її код більший на одиницю (так, літера p має в кодовій таблиці ASCII код 112, а літера q — код 113).

Це дозволяє зручно перевіряти приналежність символа до певного діапазону навіть без допомоги функцій класикації. Наприклад, наведемо оператор, який розпізнає, чи є символ, що зберігається у змінній `c` великою літерою латинського алфавіту.

```
char c;
. . .
if( ('A' <= c) && (c <= 'Z') )
  printf( "так" );
else
  printf( "ні" );
```

На жаль, існує кілька різних способів кодування символів національних алфавітів, і далеко не в усіх кодових таблицях літери кирилічної абетки розташовані за порядком. Тому для порівняння кирилічних літер показаний вище простий спосіб в загальному випадку не підходить.

8.3. Зберігання рядків в пам'яті

Ключові слова: текстовий рядок, масив символів, завершальний нуль-символ, рядкові константи.

Відмінність мови C від деяких інших мов (зокрема, Бейсіку чи Паскаля) в тому, що в C немає спеціального типу даних «рядок». Замість цього рядок *моделюється* за допомогою тих самих стандартних засобів, що вивчалися в попередніх розділах.

Текстовий рядок є лінійною послідовністю, зіставленою з окремих літер. Кожна літера (символ) в послідовності має свій номер, що відлічується від початку рядка. Тому в мові C прийнято, що текстовий рядок це масив елементів символьного типу. Таким чином, обробка тексту в мові C моделюється через обробку елементів масиву. Приклад оголошення такого масиву:

```
char str[40];
```

Найважливіша для всього подальшого розгляду особливість полягає в тому, що наприкінці рядка, після всіх «справжніх» символів повинен стояти переривач, тобто спеціальний символ, що відіграє роль ознаки кінця рядка. Це символ з числовим кодом 0.

Розглянемо приклад. Нехай дано оголошення масиву та оператори, які присвоюють значення першим чотирьом його елементам:

```
char str[10];
str[0] = 'д';
str[1] = 'а';
str[2] = 'р';
str[3] = 0;
```

Після присвоювань перші елементи масиву утворюють слово «дар». В наступному елементі масиву стоїть нуль-символ, що означає кінець тексту. Текст, що міститься в масиві, складається з 4 символів (рахуючи нуль-символ), а масив має довжину 10 елементів. Це означає, що решта 6 елементів просто не використовуються: при будь-якій обробці тексту (скажімо, при виведенні на екран) враховуються лише ті символи, що стоять від початку масиву до нуль-символа.

Увага! Отже, фактична довжина рядка, що зберігається в масиві символів, може бути меншою за зарезервований для масиву об'єм пам'яті.

В багатьох задачах є сенс навмисно резервувати для рядка більше пам'яті, ніж потрібно — це дозволяє програмі розширювати рядок. Скажімо, попередній приклад можна продовжити операторами

```
str[3] = 'у';
str[4] = 'ю';
str[5] = 0;
```

В тому ж самому масиві, де раніше зберігалося слово «дар», тепер сформовано текст «дарую», більшої довжини.

У наведеному вище прикладі текст формувався у масиві літера за літерою, присвоюванням значень кожному елементу масиву. Мова C, звичайно ж, підтримує зручніший та компактніший спосіб позначати рядки.

Рядкова константа позначається як набір символів, взятий в подвійні лапки. Прикладами рядкових констант можуть бути форматні рядки функції `printf`, що використовувалися з перших розділів цього посібника.

Рядкові константи можуть використовуватися для надання початкових значень рядкам-масивам, наприклад:

```
char str[20] = "Древляни";
```

Нарешті, мова C допускає таку додаткову зручність: не вказувати власноруч кількість елементів масиву, якщо одразу ж при оголошенні йому присвоюється початкове значення, наприклад:

```
char str[] = "Древляни";
```

В такому випадку компілятор сам порахує символи в рядку (рахуючи і завершальний нуль-символ) та створить масив точно такого розміру.

8.4. Введення-виведення рядків

Для виведення рядка на друк у функції `printf` застосовується форматний специфікатор `%s`. Наприклад, наведена нижче програма містить текстовий рядок з іменем `str`, який виводиться на екран разом з текстом форматного рядка.

```
#include<stdio.h>
int main() {
char str[] = "Вітаю!";
printf( "В␣масиві␣текст␣<%s>\n", str );
return 0;
}
```

Для введення рядків у функції `scanf` також застосовується специфікатор `%s`. Але при введенні рядків треба мати на увазі кілька важливих особливостей. Якщо не врахувати ці «підводні камені», може виникнути прикра помилка, яка зробить роботу програми непередбачуваною.

Розглянемо спершу найпростіший випадок. У наведеному нижче програмному фрагменті оголошується рядок з іменем `str`, для якого резервується 20 байт пам'яті. Це означає, що «справжніх» символів в ньому може бути не більше 19, оскільки один байт повинен займати нуль-символ. В масив `str` розміщуються символи, які вводяться з клавіатури (до натискання клавіші Enter).

```
char str[20];
scanf( "%s", str );
```

Треба звернути увагу, що `str` є ім'ям масиву, а отже покажчиком, тому ставити перед ним операцію взяття покажчика `&` непотрібно.

Уявімо, що користувач вводить не 19, а, скажімо, 30 символів. Мова C, як неодноразово наголошувалося вище, не підтримує автоматичний контроль виходу за границю масиву, тому перші 20 з введених користувачем символів розмістяться в зарезервованих комірках пам'яті в елементах масиву `str`, а решта 10 символів та завершальний нуль-символ підуть у комірки безпосередньо після масиву і отже можуть зіпсувати значення інших змінних або коди машинних команд. В обох випадках результат може бути дуже неприємним.

Увага! Введення текстового рядка — потенційно небезпечна операція: якщо застосовується просто специфікатор `%s`, залишається лише сподіватися, що користувач введе стільки символів, на скільки розрахована місткість масиву.

Проблему можна вирішити, якщо явно вказати у специфікаторі граничну довжину рядка. Довжина вказується десятковим числом між знаком `%` та літерою `s`. Так, в попередньому прикладі треба написати

```
scanf( "%19s", str );
```

Якщо користувач введе 30 символів, то лише перші 19 з них підуть в масив `str`, в останній елемент масиву буде занесено нуль-символ, а решта набраних користувачем на клавіатурі символів буде проігноровано цим викликом `scanf` — вони залишаться в буфері та, можливо, будуть введені наступними викликами.

Увага! При введенні текстового рядка треба обов'язково обмежувати максимальну довжину рядка у специфікаторі.

Треба також мати на увазі, що функція `scanf` зі специфікатором `%ns` (де n — число), якщо набрати на клавіатурі текст з кількох слів, розділених пробілами, вводить лише символи до першого пробілу. Для того, щоб вводити тексти з пробілами, треба або використовувати більш витончені можливості функції `scanf`, які в цьому посібнику розглядатися не будуть, або інші функції, наприклад `fgets`[1].

Функція `fgets`, дозволяє вводити довільний текст, що містить пробіли та знаки табуляції, в межах заданої довжини. Функція має три аргументи. Першим аргументом є покажчик `s` на масив символів, в якому функція має розмістити введений користувачем текст. Другий аргумент, ціле число `n` — гранична довжина тексту, зазвичай це число дорівнює довжині масиву `s`. Якщо користувач вводить надто багато літер (більше, ніж `n`), то функція `fgets` зчитує лише перші `n-1` символів та заносить їх до масиву `s`, а решта символів залишаються в так званому буфері, вони будуть прочитані при наступних викликах функцій `fgets` або `scanf`.

Сенс третього аргументу функції `fgets` в даному розділі пояснювати не будемо, оскільки відповідні поняття викладаються в розділі 10. Поки що обмежимося вказівкою, що для введення тексту з клавіатури до третього аргументу потрібно передавати значення `stdin`.

Отже, повністю конструкція для введення тексту, що може містити пробіли, виглядає так, як показано нижче.

```
#define N 40
...
char str[ N ];
fgets( str, N, stdin );
printf( "Ви␣ввели␣текст:\n%s\n", str );
```

8.5. Функції для обробки рядків

Ключові слова: довжина рядка, копіювання рядків, порівняння рядків.

Всі дії над рядками реалізовано у вигляді функцій, оголошення яких зібрано в заголовочному файлі `string.h`. Ці функції мають один чи більше аргументів типу покажчиків на рядки, які належить обробити.

Функція `strlen` приймає один аргумент типу покажчика на рядок та повертає його довжину (ціле число). Функція рахує лише «справжні» символи, тобто крім завершального нуль-символу. Приклад:

```
1 #include<string.h>
2 int main() {
3 char str[80];
4 int n;
5 printf( "Введіть␣слово␣" );
6 scanf( "%79s", str );
7 n = strlen( str );
```

[1]Існує широко відома функція `gets`, яка дозволяє вводити текст, що містить пробіли, але користуватися нею не рекомендується, оскільки вона не підтримує граничну довжину тексту, що вводиться, і тому ненадійна та потенційно небезпечна.

```
8 printf( "В␣цьому␣слові␣%d␣символів", n );
9 return 0;
10 }
```

Не зайвим буде нагадати, що число 80 в оголошенні масиву — це не довжина рядка, а об'єм зарезервованої пам'яті. Рядок може містити менше симовлів, ніж зарезервовано.

Для копіювання рядків слугує функція `strcpy`. Вона приймає два аргументи типу покажчиків на рядки: перший — покажчик на рядок-приймач (куди копіювати), другий — покажчик на рядок-джерело (звідки копіювати). Приклад:

```
1 #include <stdio.h>
2 int main() {
3 char s1[] = "Князь␣Святослав␣Ігоревич";
4 cahr s2[80] = "Хазари";
5 strcpy( s2, s1 );
6 printf( "Тепер␣в␣рядку␣2:␣%s\n", s2 );
7 return 0;
8 }
```

Другий рядок має зарезервовану граничну довжину 80 байт (79 символів плюс нуль-символ), з них спершу реально зайнято лише 6 символів. Функція `strcpy` копіює символ за символом з рядка `s1` до області пам'яті `s2`. Символи копіюються на початок області, тому затирають те, що було в масиві `s2` раніше. Отже, після виконання функції `strcpy` в двох областях пам'яті, `s1` та `s2`, будуть міститися однакові послідовності символів «Князь Святослав Ігоревич» (а «Хазари» зникають безслідно). Цю дію і називають копіюванням рядків.

Цікаво буде порівняти копіювання рідків з присвоюванням цілих чисел. Розглянемо дві змінні, `x` та `y` типу **`int`**. Для того, щоб в змінній `x` отримати точну копію значення, яке міститься в змінній `y`, застосовується оператор присвоювання вигляду `x=y`. Натомість, для рядків неможна застосовувати операцію присвоювання. Якщо рядки оголошено як масиви, операція присвоювання взагалі не має сенсу (ім'я масиву є покажчиком-константою, а константі неможна присвоїти нове значення). Якщо ж рядки оголошено як змінні-покажчики, як в наведеному нижче тексті,

```
char *p = "Київська", *q = "Русь";
```

то операція присвоювання виконається, але її результатом буде зовсім не копіювання вмістів рядків. Змінні `p` та `q` є покажчиками. Тобто ці змінні містять не самі текстові рядки, а адреси в пам'яті, по яких розміщено рядки. Перед присвоюванням змінна `p` містить адресу області пам'яті, в якій лежить послідовність літер «Київська», а змінна `q` містить адресу рядка «Русь». Операція присвоювання бере значення змінної `q` (адресу другого рядка) та копіює його в змінну `p`. Тепер обидві змінні вказують на одну і ту саму область пам'яті, де лежать символи «Русь». Іншими словами, операція присвоювання не створила другу копію рядка, а просто зробила другий покажчик на той самий оригінал. При цьому, до речі, рядок «Київська» так і залишається в пам'яті на тому ж місці, але покажчик на нього втрачено.

Функція з'єднання рядків `strcat` має два аргументи типу покажчиків на символьні рядки: перший — рядок-приймач (до якого рядка дописувати) та другий — рядок-джерело (що дописувати). Функція дописує другий рядок до кінця першого. Розглянемо, наприклад, програмний фрагмент

```
char s1[80] = "Древлянська␣";
char s2[] = "земля";
strcat( s1, s2 );
```

Тут для першого рядка зарезервовано місце з запасом. Наприкінці першого рядка стоїть пробіл. Виклик функції `strcat` копіює символ за символом з рядка `s2` в область пам'яті `s1`, в те місце, де закінчувався його попередній зміст. Отже, тепер в масиві `s1` містяться символи «Древлянська земля» та нуль-символ в кінці.

Функція `strcmp` має два аргументи типу покажчиків на рядки. Функція порівнює рядки та повертає результат порівняння, ціле число: 0 — якщо рядки співпадають, деяке від'ємне

число — якщо перший рядок передує в словниковому порядку, та додатнє число — якщо передує другий рядок.

Під словниковим порядком розуміють порядок, в якому слова розташовують у словнику за абеткою. Нехай дано два слова. Спочатку порівнюємо першу літеру. Якщо перша літера першого слова стоїть в абетці раніше, ніж перша літера другого слова, то перше слово передує другому в словниковому порядку. Наприклад, слово «Житомир» передує слову «Київ». Якщо перші літери двох слів співпадають, то порівняння проводять за другою літерою, і так далі. Наприклад, слово «Життя» йде в словниковому порядку після слова «Житомир», оскільки перші три літери в них однакові, а на четвертому місці в першому слові стоїть літера «т», яка за абеткою йде пізніше за літеру «о».

Рядки треба порівнювати лише за допомогою функції `strcmp`, неможна порівнювати їх операцією `==`. Справді, рядки обробляються як масиви символів, а масиви, в свою чергу, є покажчиками. Тому порівняння вигляду `s1==s2` просто порівняє два покажчики, тобто перевіряє, чи вказують вони на одну й ту саму адресу. Натомість функція `strcmp` порівнює не адреси, а самі послідовності символів, що лежать в пам'яті за цими адресами.

В наведеному нижче прикладі програма запитує у користувача пароль, порівнює його з закладеним в текст програми еталоном та завершує роботу, якщо пароль користувача неправильний (не збігається з еталоном), або продовжує роботу, якщо пароль правильний.

```
1  #include <string.h>
2  int main() {
3  char pswd[] = "слово";
4  char buf[20];
5  printf( "Введіть␣свій␣пароль␣" );
6  scanf( "%s", buf );
7  if( strcmp( buf, pswd ) != 0 ) {
8    printf( "Доступ␣заборонено\n" );
9    return 0;
10 }
11 /* далі нормальна робота програми */
12 . . .
13 }
```

Наостанок неможна не сказати про надзвичайно корисні функції `sprintf` та `sscanf`. Ці функції мають першим аргументом покажчик на деякий текстовий рядок, далі йдуть такі ж аргументи, як і у функцій `printf` та `scanf` відповідно (тобто форматний рядок і довільна кількість значень для першої функції або покажчиків для другої). На відміну від функцій `printf`, функція `sprintf` «друкує» значення замість екрану в текстовий рядок, а функція `sscanf` вводить значення з рядка замість клавіатури. Наприклад:

```
char str[80];
char month[] = "вересень";
int day = 12;
sprintf(str, "місяць␣%s,␣число␣%d", month, day);
```

Після виконання функції `sprintf` в масиві `str` сформується послідовність символів `місяць: вересень, число: 12` (з завершальним нуль-символом наприкінці).

8.6. Рядки в динамічій пам'яті

В наведених вище прикладах для розміщення рядків використовувалися масиви з автоматичним виділенням пам'яті, які мають заздалегідь жорстко заданий розмір. В цьому розділі розглянемо роботу з рядками, коли пам'ять виділяється динамічно, а потрібний обсяг програма визначає по ходу роботи.

Оскільки масив та покажчик на перший елемент масиву в мові C одне й те саме, спосіб обробки рядків за допомогою функцій не залежить від того, автоматично чи динамічно виділено пам'ять для цих рядків. Якщо в програмі є змінна `p` типу покажчика на символ **`char*`**, і

вона вказує на початок текстового рядка, то її можна передавати як аргумент в усі розглянуті вище функції.

Єдина справжня відмінність полягає в тому, що програма повинна обчислити довжину рядка та виділити для нього пам'ять. Нижче наведено приклад програми, яка динамічно виділяє область пам'яті, в якій з'єднує три рядки. Для того, щоб розмір виділеної області в точності відповідав потребі, програма спочатку вимірює довжину кожного з трьох рядків.

```
#include <string.h>
int main() {
char s1[] = "Слово␣";
char s2[] = "о␣полку␣";
char s3[] = "Ігоревім";
char *p;
int n1, n2, n3;
n1 = strlen( s1 );
n2 = strlen( s2 );
n3 = strlen( s3 );
p = (char*) malloc( n1+n2+n3+1 );
strcpy( p, s1 );
strcat( p, s2 );
strcat( p, s3 );
printf( "%s\n", p );
printf( "тут␣%d␣символів\n", strlen(p) );
free( p );
return 0;
}
```

Далі програма виділяє область пам'яті, розмір якої дорівнює сумі довжин трьох рядків плюс один байт. Останній потрібен, щоб розмістити завершальний нуль-символ. Перш за все, у виділену область копіюється перший рядок. Далі до його кінця програма дописує другий рядок. Потім до кінця рядка, який вийшов в результаті, дописує третій рядок.

Виклик функцій `printf` та `strlen` зі змінною `p` в якості аргументу ілюструє, що покажчик на динамічну область пам'яті, в якій містяться символи, може оброблятися так само, як, наприклад, масив `s1`.

8.7. Підрахунок числа пробілів

Розглянемо кілька прикладів типових задач обробки текстової інформації. Нехай потрібно в тексті, що ввів користувач, підрахувати кількість пробілів. Щоб не ускладнювати програму зайвими подробицями, припустимо, що довжина тексту, що обробляється, не перевищує певної межі, скажімо, 100 символів. Тоді логіку роботи програми можна описати такою схемою:

1. Ввести текст з обмеженням по довжині;
2. Встановити значення лічильника пробілів в 0;
3. Перебрати всі по порядку символи введеного тексту і для кожного перевірити, чи не є він пробілом. Якщо є, то збільшити лічильник на 1.

Зрозуміло, що після перебору всіх символів значенням лічильника і буде кількість всіх пробілів.

Далі треба вирішити, як організувати перебір всіх символів текстового рядка. Найпростіший та очевидний варіант: спочатку знайти кількість всіх символів в рядку за допомогою функції `strlen`, нехай отримаємо число `n`, а потім змінювати в циклі значення індексу `i` в межах від 0 до `n-1`; нарешті, `i`-й символ тексту є просто `i`-м елементом масиву. Отже, маємо програмний текст:

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

```
3 #define N 100
4 int main() {
5 char str[ N ];
6 int m, n, i;
7 fgets( str, N, stdin );
8 m = strlen( str );
9 n = 0;
10 for( i = 0; i < m; ++i )
11   if( str[i] == '␣' ) ++n;
12 printf( "Текст␣містить␣%d␣пробілів", n );
13 return 0;
14 }
```

Ця програма правильна, але недосконала. Для того, щоб порахувати довжину рядка, функція `strlen` продивляється його від першого символа до останнього (поки не знайде нуль-символ), а потім алгоритм підрахунку пробілів знов продивляється весь рядок, тобто алгоритм працює в два проходи. Можна досягти економії часу, якщо поєднати два проходи в один.

Нехай алгоритм перебирає символи від першого й далі. Якщо поточним символом є пробіл, до лічильника пробілів додається 1. Якщо поточним символом є нуль-символ, алгоритм завершує роботу. Відповідний програмний текст показано нижче:

```
1 #include <stdio.h>
2 #define N 100
3 int main() {
4 char str[ N ];
5 int n, i;
6 fgets( str, N, stdin );
7 n = 0;
8 for( i = 0; str[i] != 0; ++i )
9   if( str[i] == '␣' ) ++n;
10 printf( "Текст␣містить␣%d␣пробілів\n", n );
11 return 0;
12 }
```

Запитання

1. Скласти програму, яка за один прохід визначає, скільки в тексті знаків питання та знаків оклику
2. Скласти програму, яка за один прохід визначає, скільки в тексті знаків арифметичних операцій +, -, *, /.
3. Скласти фнкцію, яка, маючи два аргументи — покажчики на рядки (припускаємо, що в другому рядку всі символи рязні) — обраховує, скільки разів в першому рядку трапляються символи, які є в другому рядку.

8.8. Зсув та вставка

Напишемо функцію, яка отримує покажчик на текстовий рядок та ціле невід'ємне число n і циклічно пересуває всі символи рядка на n позицій ліворуч. Це означає, що символ, який стояв в $n+1$-й позиції, стане першим, $n+2$-й стане дргим і т.д, а перші n символів рядка перейдуть на його кінець. Наприклад, рядок «Програмування» після такого перетворення з $n=5$ перетвориться на «амуванняПрогр».

Можна запропонувати кілька варіантів алгоритму, тут розберемо найочевидніший. Створимо буфер потрібного розміру та збережемо в ньому перші n символів рядка, що перетворюється. Потім символи, починаючи з $n+1$-го і до кінця рядка, зсунемо на n позицій ліворуч. Нарешті, вміст буфера перенесемо знов в рядок, що обробляється, але вже в його кінець.

Буфер потрібно створювати динамічно, оскільки його розмір — число n — не може бути відомим заздалегідь. Оскільки буфер потрібен лише на час роботи алгоритму, наприкінці його треба знищити. Остаточно текст функції має такий вигляд:

```
void str_lshift( char *s, int n ) {
int i, k;
char *b;
b = (char*) malloc( n * sizeof(char) );
for( i = 0; i < n; ++i )
  b[i] = s[i];
k = strlen( s );
for( i = n; i < k; ++i )
  s[i-n] = s[i];
for( i = 0; i < n; ++i )
  s[k-n+i] = b[i];
free( b );
}
```

Розберемо приклад функції, яка в один рядок на задану позицію вставляє інший рядок. Наприклад, вставка в рядок «князь Святослав» на позицію 6 рядка «слов'янський » дасть результат «князь слов'янський Святослав». Функція повинна мати три аргументи: покажчик на рядок, до якого буде здійснюватися вставка, покажчик на другий рядок, який буде вставлено в перший, та ціле число, позиція символа, з якого починається вставка. При цьому вважаємо, що перший покажчик показує на область пам'яті, в якій не лише міститься перший рядок, але й залишається достатньо вільного місця для вставки другого рядка.

```
void str_insert( cgar *s, char *t, int k ) {
int i, m, n;
m = strlen( s );
n = strlen( t );
for( i = m; i >= k; i-- )
  s[i+n] = s[i];
for( i = 0; i < n; ++i )
  s[i+k] = t[i];
}
```

Спочатку ця функція має пересунути в рядку `s` символи від `k`-го до останнього на `n` позицій праворуч, де `n` — довжина рядка, який вставляється. Звертаємо увагу, що ці символи треба обробляти у зворотному порядку, останній символ рядка треба пересувати першим. В іншому випадку `k`-й символ рядка, скопійований на `n` позицій вперед, зіпсував би `k+n`-й символ.

Запитання

1. Написати функцію, яка робить циклічний зсув рядка на n символів праворуч.
2. Написати функцію, яка робить циклічний зсув рядка ліворуч (праворуч) на один символ, максимально спростивши алгоритм.
3. Написати програму, яка запитує у користувача рядок тексту (довжиною не більше 79 символів, щоб вміститися в один рядок екрану) та демонструє на екрані рядок, який біжить по екрану вліво: через кожні 0,1 с. рядок зсувається на 1 символ ліворуч, а той символ, що добіг лівого краю екрана, знов з'являється з правого краю. Підказка 1: скористатися спеціальним символом «`\r`», друк якого пересуває курсор на початок поточного рядка екрану так, що наступні символи будуть затирати ті, які були надруковані в цьому рядку раніше. Підказка 2: для затримки виконання програми на заданий проміжок часу в реалізаціях мови C під операційною системою MS DOS слугує функція `delay`, аргументом якої є ціле число, кількість мілісекунд, на яку треба зробити затримку. Під операційною системою UNIX слід використовувати функцію `sleep`.
4. Написати функцію, яка видаляє з рядка фрагмент довжини l, починаючи з позиції k.

5. Написати функцію, яка, маючи два аргументи, покажчики на рядки, видаляє з першого рядка всі символи, які містяться в другому рядку.

8.9. Підрахунок числа слів

Ключові слова: скінченний автомат, стан, перехід.

Задача. Скласти програму, яка підраховує кількість слів у введеному користувачем тексті. Словом вважається послідовність літер латинського алфавіту. Наприклад, у наведеному нижче тексті 7 слів:

```
Switch 01:07 is on, turn KWELL+TRIFF... success!
```

Як видно, між словами можуть стояти пробіли, цифри, знаки пунктуації тощо. Словом вважається і одна латинська літера, що стоїть між будь-якими нелітерними символами. У проміжку між двома сусідніми словами може стояти не лише один пробіл, а будь-яка кількість пробілів.

Перш за все, треба винайти основну ідею алгоритму. Будемо продивлятися текст від початку послідовно, символ за символом. На кожному кроці будемо оглядати лише один символ (назвемо його поточним символом), на кожному наступному кроці будемо переходити на один символ праворуч. Якщо попередній символ був не літерою, а поточний є літерою, то значить поточний символ є початком слова, тоді запам'ятаємо, що алгоритм *увійшов у слово*. Якщо на наступних кроках алгоритм знаходиться в слові і бачить, що поточний символ знов є літерою, він просто переходить до наступного символу, пам'ятаючи, що він досі знаходиться у слові.

Якщо алгоритм знаходиться у слові та на черговому кроці бачить, що поточний символ є не літерою, це значить, що знайдено кінець слова. В такому випадку треба додати одиницю до лічильника слів та запам'ятати, що віднині алгоритм знаходиться не в слові. Якщо алгоритм знаходиться не в слові та бачить, що черговий символ є не літерою, то він, продовжуючи пам'ятати, що знаходиться не в слові, переходить до наступного символу. Нарешті, якщо алгоритм знаходиться не в слові та бачить в поточній позиції літеру, він знов запам'ятовує, що потрапив у слово.

Опишемо цей принцип роботи більш строго. В основу опису покладемо широко розповсюджену в комп'ютерних науках метафору *скінченного автомату*. В кожен момент часу автомат може перебувати в одному з кількох *станів*. На кожному кроці поведінка автомата визначається двома чинниками: його поточним станом та черговим символом, який він сприймає з вхідного тексту. Дія автомата на кожному кроці — це *перехід* у новий стан та, можливо, деякі операції над даними.

В нашому випадку автомат має два стани, які назвемо *слово* та *проміжок*. Скористаємося тим, що кінець тексту відмічено символом з числовим кодом 0, який, звичайно ж, не є літерою. Робота автомата визначається такими правилами:

1. Спочатку алгоритм перебуває в стані *проміжок*, поточним є перший символ тексту, лічильнику слів присвоюється початкове значення 0.
2. Залежно від поточного стану та поточного символу обрати один з чотирьох варіантів:
   - Якщо поточний символ є літерою, а станом є *проміжок*, то змінити стан на *слово*;
   - Якщо поточний символ є літерою, а станом є *слово*, то нічого не робити;
   - Якщо поточний символ є не літерою, а станом є *проміжок*, то нічого не робити;
   - Якщо поточний символ є не літерою, а станом є *слово*, то змінити стан на *проміжок* та додати одиницю до лічильника слів.
3. Якщо поточний символ є нуль-символом, то завершити роботу алгоритму.
4. Перейти до наступного символу та продовжити роботу з кроку 2.

Зазначимо, що хоча два з чотирьох наведених варіантів виглядають зайвими (в цих двох випадках алгоритм повинен нічого не робити), їх варто включити в опис алгоритму — для логічної повноти.

В підсумку маємо такий текст програми підрахунку слів:

```
#include <stdio.h>
#define N 100
int main() {
char str[ N ];
int n = 0; /* лічильник слів */
int i; /* номер поточного символу */
int state = 0; /* стан */
/* 0 - проміжок, 1 - слово */
fgets( str, N, stdin );
while( 1 ) {
  if( isalpha(str[i]) && (state==0) )
    state = 1;
  if( isalpha(str[i]) && (state==1) )
    { }
  if( !isalpha(str[i]) && (state==0) )
    { }
  if( !isalpha(str[i]) && (state==1) ) {
    state = 0;
    ++n;
  }
}
printf( "Текст␣містить␣%d␣слів\n", n );
return 0;
}
```

Цей нескладний приклад ілюструє «в мініатюрі» типові принципи роботи компіляторів.

Запитання

1. Скласти програму, яка підраховує у введеному користувачем тексті кількість чисел у десятковій системі. Числом вважаємо послідовність цифр 0–9.
2. Скласти програму, яка за один прохід підраховує в тексті кількість слів та кількість чисел.

8.10. Пошук підслова

Це одна з найтиповіших та найважливіших на практиці задач символьної обробки. Дано текст — достатньо довгу послідовність символів та зразок — відносно коротку послідовність символів. Треба знайти в тексті перше входження зразка (видавши номер символу, в якому починається входження) або встановити, що зразок в тексті не зустрічається.

Сформулюємо основну ідею алгоритму. Позначимо текст, в якому ведеться пошук, через t, а зразок — через p. Нехай довжина тексту t становить n, а довжина зразка p — m символів. Будемо переглядати текст t символ за символом, нехай i є поточною позицією перегляду (за загальним правилом мови C вважаємо, що нумерація починається з 0).

Для кожного значення i треба перевірити, чи не починається в i-й позиції тексту t підслово, що співпадає зі зразком p. Для такої перевірки треба перебрати по черзі всі літери слова p (нехай j є номером поточної літери зразка) та порівняти j-у літеру зразка з $(i+j)$-ю літерою тексту.

Якщо для всіх $j = 0, ..., m-1$ така рівність виконується, то це значить, що алгоритм завершується, входження зразка у текст знайдено, входження починається з i-ї позиції. Якщо

ж при хоча б одному значенні j вони не співпадають, значить в i-й позиції тексту немає входження зразка, і тому треба переходити до наступного значення i.

Якщо поточна позиція в тексті досягла значення $i = n - m + 1$ (тобто від поточної позиції до кінця слова залишилося менше символів, ніж містить зразок), то, зрозуміло, немає сенсу продовжувати пошук — алгоритм завершується, встановивши, що входжень даного зразка у тексті немає.

Щоб продемонструвати справді гарний стиль програмування, оформимо цей алгоритм у вигляді функції. Функції дамо ім'я `findSubword` (дослівно — знайти підслово). Її аргументами, очевидно, повинні бути два покажчики на текстові рядки: `t` та `p`. Функція повертає ціле число: якщо входження зразка у текст знайдено, то номер позиції, з якої воно починається; якщо входжень не існіє, то число -1.

```
1  int findSubword( char *t, char *p ) {
2  int i, j, m, n;
3  m = strlen( p );
4  n = strlen( t );
5  for( i = 0; i <= n-m; ++i ) {
6    for( j = 0; j < m; ++j )
7      if( t[i+j] != p[j] ) break;
8    if( j == m ) return i;
9  }
10 return -1;
11 }
```

Оператор **break** в рядку 7 виконується лише тоді, коли при спробі накласти зразок на текст знайдено розбіжність в одній літері. Цей оператор перериває внутрішній цикл (по змінній `j`, перебір літер зразка). Перевірка в рядку 8 дає позитивний результат лише тоді, коли внутрішній цикл пройшов по всіх значеннях `j`, а це можливо лише тоді, коли жодного разу не знайдено розбіжності між літерою тексту та літерою зразка. При позитивному результаті оператор **return** завершує виконання функції та повертає значення змінної `i`. Негативний результат перевірки свядчить про те, що зразок не співпав з поточним фрагментом тексту, отже алгоритм повинен перейти до наступної ітерації зовнішнього циклу.

Для демонстрації роботи побудованої функції складемо програму, яка запитує у користувача текст і шукає у ньому слово «слава».

```
1  #include <stdio.h>
2  #include <string.h>
3
4  /* прототип функції */
5  int findSubword( char *t, char *p );
6
7  #define N 100
8
9  int main() {
10 char theText[N];
11 char thePattern[] = "слава";
12 int k;
13 k = findSubword( theText, thePattern );
14 printf( "входження␣слова␣%s:␣", thePattern );
15 if( k != -1 )
16   printf( "починається␣з␣%d-ї␣позиції\n", k );
17 else
18   printf( "немає\n" );
19 return 0;
20 }
21 /* сюди вставити означення
22 функції, наведене вище */
```

Цей алгоритм гарантує правильний результат, але недосконалий, оскільки має невисо-

ку швидкість роботи. Існує велика кількість більш складних алгоритмів пошуку підслова, в яких вдається досягти значного виграшу в швидкості [4]. Крім того, описаний вище алгоритм забезпечує пошук лише найпростішого зразка — наперед заданого підслова, тоді як більш загальні алгоритми дозволяють шукати складні зразки (наприклад, підслово, що починається з літери «а», має рівно шість приголосних та закінчується на голосну). Розгляд таких алгоритмів, надзвичайно важливих через масову потребу в пошуку інформації у величезних масивах глобального інформаційного середовища, виходить далеко за межі курсу основ програмування.

8.11. Підсумковий огляд

Тип **char** слугує для роботи з короткими (8 двійкових розрядів) цілими числами, до яких належать і числові коди символів. Кодові таблиці задають, який символ яким числом зображується. Існують технічні труднощі через те, що немає єдиної загальносвітової кодової таблиці, в різних системах використовується декілька відмінних між собою кодувань.

Мова C не містить спеціального типу даних «текст» або «рядок». Натомість робота з текстовою інформацією моделюється за допомогою масивів символів. Робота з рядками ґрунтується на угоді про спосіб їх зберігання в пам'яті: символ з кодом 0 позначає кінець рядка. Довжина рядка (кількість символів до завершального нуля) може бути меншою за об'єм пам'яті, виділений для масиву — це дозволяє залишати в пам'яті вільне місце, щоб потім розширювати рядок.

Стандарт мови C визначає широкий набір функцій, що реалізують найтиповіші операції над рядками: дізнатися довжину рядка, скопіювати рядок, порівняти два рядки за лексикографічним порядком, а також багато інших.

Символьна обробка становить окрему велику галузь комп'ютерних наук, особливе значення мають синтаксичний аналіз та компіляція. Математичні моделі та методи символьної обробки вивчаються в окремих курсах.

Хоча мовою C в принципі можливо запрограмувати будь-яку задачу символьної обробки, загалом мова не надто добре підходить для таких задач. Незручність полягає в тому, що програмісту доводиться витрачати велику частину часу та зусиль не на власне прикладну задачу, а на рутинні допоміжні операції, такі як виділення пам'яті для рядків, перевірку того, чи вистачає зарезервованого для рядка місця тощо. Але, порівняно зі спеціалізованими мовами символьної обробки, мова C дозволяє писати програми з максимальною ефективністю, тобто надзвичайно швидкі та невимогливі до пам'яті.

Розділ 9

Структурні типи

9.1. Основні поняття

Ключові слова: структурний тип, структурний об'єкт, поле (член) структури, операція звертання до поля структури.

Часто на практиці виникає бажання зберігати деякі дані не окремо, кожну в своїй змінній, а однією «пачкою». Це буває зручно тоді, коли кілька значень настільки тісно пов'язані між собою за сенсом задачі, що всякий раз, коли нам потрібно звернутися до одного з цих значень, майже неодмінно поряд виникне потреба звернутися і до інших.

Наприклад, комплексне число зручно уявляти як пару дійсних чисел — відповідно, дійсної та уявної частини. З одного боку, обидва ці числа є окремими значеннями, але при майже будь-яких обчисленнях вони виступають разом і за сенсом задачі утворюють одне логічне ціле. Наприклад, можна говорити і про весь об'єкт «комплексне число z», і про «уявну частину b комплексного числа z» — другий об'єкт є складовою частиною першого об'єкту.

Облікова картка працівника містить в собі такі значення, як прізвище (текстовий рядок), ім'я, по батькові (також текстові рядки), рік народження (ціле число), зарплата (дійсне число). Ці значення також є водночас і самостійними об'єктами даних, і складовими одного цілого.

Загалом, для людини, яка розмірковує про прикладну задачу, природним є поєднувати елементарні дані у смислові блоки. В мові C є спеціальний засіб, що підтримує такий мислительний прийом.

Структура є таким об'єктом даних, який складається з інших іменованих даних певних типів (їх називають *полями* або *членами* структури). Іншими словами, структура це об'єкт, який містить в собі кілька змінних, в тому числі різних типів.

Перед будь-яким використанням структур треба оголосити структурний тип. Оголошення структурного типу має вигляд

```
struct ім'я_структурного_типу {
  тип_поля ім'я_поля;
  ...
  тип_поля ім'я_поля;
};
```

Слово `struct` є службовим словом мови C. Ім'ям структурного типу та іменами полів можуть бути будь-які ідентифікатори. Типами полів можуть бути будь-які стандартні типи мови C або типи, створені самим програмістом та оголошені в тексті програми раніше.

Наприклад, оголосимо типи даних, що моделюють точку на площині та особу:

```
struct tPoint {
  double x;
  double y;
};

struct tPerson {
  char name[20];
  char surname[20];
  int yearOfBirth;
};
```

Ці оголошення означають, що кожен об'єкт структурного типу `tPoint` має в собі два поля дійсного типу з іменами `x` та `y`; кожен об'єкт структурного типу `tPerson` містить в собі поле

`name`, що є масивом з 20 символів, поле `surname`, що також є масивом з 20 символів, та поле `yearOfBirth` цілого типу.

Після того, як оголошено структурний тип, можна оголошувати змінні структурного типу — тобто змінні, значеннями яких будуть структури. Оголошення змінних структурного типу принципово не відрізняється від розібраних вище оголошень змінних простих типів. Треба лише мати на увазі, що тип змінної треба обов'язково записувати словосполученням **struct** `ім'я_структурного_типу`, а не одним лише словом[1] `ім'я_структурного_типу`. Нижче наведено форму оголошення змінних структурного типу.

```
struct ім'я_структурного_типу ім'я_змінної;
```

Наприклад, скориставшися наведеними вище прикладами оголошення структурних типів, оголосимо змінні `A` та `B` типу `tPoint`.

```
struct tPoint A, B;
```

Після такого оголошення в програмі виникає об'єкт `A`, всередині якого містяться змінні, або поля `x` та `y`, та об'єкт `B`, в якому є свої поля з такими ж іменами. Зрозуміло, що *«поле `x` зі структурного об'єкта `A`»* та *«поле `x` зі структурного об'єкта `B`»* — це зовсім різні змінні, які розташовані в різних комірках пам'яті та мають кожна своє значення (так само, як «оцінка студента М. з програмування» та «оцінка студента Н. з програмування» — це різні величини).

Тепер, оголосивши змінні структурного типу, їх потрібно обробляти, робити з ними в програмі певні дії. Основним різновидом операції над структурою є *виділити в структурі певне поле та обробити його як звичайну змінну.* В наведеному прикладі це може бути «присвоїти полю `x` структури `A` значення 0», або «обчислити суму квадратів значень полів `x` та `y` структури `B`».

Така операція в мові C називається звертанням до поля (або члена) структури та позначається крапкою, ліворуч від якої стоїть вираз структурного типу, а праворуч — ім'я поля: `структура.ім'я_поля`. Наприклад

```
A.x = 0;
r = B.x * B.x + B.y * B.y;
```

Запитання

1. Для чого призначені структурні типи даних?
2. Як оголошується структурний тип?
3. Як оголошується змінна структурного типу?
4. Що таке поле структурного об'єкту?
5. Як звернутися до поля структурного об'єкту?
6. Записати оголошення структурного типу «студент» з полями «ім'я», «прізвище», «рік народження», «рік вступу до університету», «спеціальність».
7. Оголосити дві змінні структурного типу «студент» (див. попереднє завдання). Скласти програму, яка в поля першої змінної присвоює ваші особисті дані, а значення полів другої структурної змінної вводить з клавіатури.

9.2. Додаткові можливості

Ключові слова: ініціалізація змінних структурного типу, присвоювання структур, порівняння структур, скорочені імена для типів, **typedef**.

Вище пояснювалося, що при оголошенні змінних простих типів можна одразу ж надавати їм початкові значення:

[1] Це правило мови C. В мові C++ можна обмежитися тим, щоб вказати саме лише ім'я структурного типу.

```
int k = -1;
double m = 2.07;
```

Не є винятком і змінні структурних типів. Особливість полягає лише в тому, що кожен структурний об'єкт є набором з кількох значень. Щоб позначити це, застосовуються фігурні дужки, наприклад:

```
struct tPoint theCenter = { 0, 0 },
  M = { 2.18, 8.317 };
struct tPerson theStudent = {
  "Туманов",
  "Святослав",
  1991
};
```

Змінним структурного типу можна присвоювати значення інших структур того ж типу:

```
struct tPoint A = {1, 4}, B;
B = A;
```

При цьому, зрозуміло, кожне поле зі структури `A` копіюється у відповідне поле структури `B`. Отже, наведене вище присвоювання дає такий самий результат, як і пара операторів

```
B.x = A.x;
B.y = A.y;
```

Структури в мові C не можна порівнювати, тобто спроба застосувати до них такі операції, як `==`, `!=`, `>` та подібні їм, компілятор вважає синтаксичною помилкою, як у фрагменті нижче:

```
struct tPoint A, B;
. . .
if( A == B ) /* помилка !!! */
  printf( "співпадають" );
else
  printf( "різні" );
```

Якщо виникає потреба в програмі порівнювати об'єкти структурного типу, програмісту потрібно в явному вигляді записати порівняння кожного поля:

```
if( (A.x == B.x) && (A.y == B.y) )
  printf( "співпадають" );
else
  printf( "різні" );
```

Варто також згадати про корисну можливість мови C. Вище було сказано, що в мові C при кожній згадці структурного типу треба використовувати повне ім'я разом зі словом **struct**, що може виявитися обтяжливим. Мова C дозволяє запроваджувати скорочення для довгих позначень типів.

Означення скороченого імені типу виглядає[1] так:

```
typedef довгий_опис_типу_даних коротке_ім'я;
```

Далі в програмі після такого оголошення можна без обмежень використовувати коротке ім'я типу даних — компілятор автоматично «подумки» підставить на його місце довгий опис. Треба мати на увазі, що оголошення **typedef** не створює новий типу, а лише приписує нове ім'я такому типу, який можна задати довгим описом.

Наведемо спочатку приклади використання **typedef**-оголошень, не пов'язані зі структурними типами. Як відомо, до типу **int** можуть приєднуватися *модифікатори*, такі як **long** (довге — збільшена розрядна сітка дозволяє зберігати більший діапазон чисел) та **unsigned** (беззнакове — діапазон такого типу складає не від $-N/2$ до $N/2-1$, а від 0 до $N-1$). Тоді

[1] Виклад доволі складної мови C для початківців приречений на сильні спрощення. Засіб **typedef** має набагато різноманітніші можливості, які в ознайомчому курсі розглядати передчасно.

кожного разу, коли потрібно оголосити змінну чи функцію беззнакового довгого цілого типу довелося б писати так:

```
unsigned long int k;
unsigned long int fff();
void ggg( unsigned long int x );
```

За допомогою засобу **typedef** стає можливим скоротити це позначення:

```
typedef unsigned long int ulint;
ulint k;
ulint fff();
void ggg( ulint x );
```

В першому рядку словосполучення **unsigned long int** є довгим описом типу даних, слово `ulint` є скороченим ім'ям, яке придумали ми. В усіх подальших рядках людина пише коротке ім'я `ulint`, але компілятор все одно «побачить» там довгий опис **unsigned long int**.

Особливо широке застосування засіб **typedef** знайшов саме для структурних типів. Розглянемо приклад:

```
struct tagPoint {
  double x;
  double y;
};
typedef struct tagPoint tPoint;
tPoint A = {1, 5}, B;
```

Спочатку оголошено структурний тип, повним ім'ям якого є словосполучення[1] **struct** `tagPoint`. Далі **typedef**-оголошення каже компілятору, що всюди далі слово `tPoint` повинно означати **struct** `tagPoint`. Нарешті, програміст може оголошувати змінні, використовуючи в якості імені типу одне слово `tPoint`.

Насправді мова С дозволяє зробити текст програми ще коротшим, поєднавши оголошення структурного типу із запровадженням скороченого імені:

```
typedef struct tagPoint {
  double x;
  double y;
} tPoint;
```

Запитання

1. Як позначається надання початкового значення змінній структурного типу? Створити будь-який структурний тип (на власний розсуд), оголосити дві змінні цього типу з наданням початкових значень.
2. Нехай `A` та `B` — дві змінні структурного типу `tPoint`. Обґрунтувати, які з цих виразів правильні, а які не мають сенсу в мові С: `A<B`, `A==B`, `A=B`, `A!=B`
3. Записати оголошення структурного типу `tTriple` з трьома полями цілого типу: `a`, `b`, `c`. Оголосити дві змінні цього типу, `P` та `Q`, одній з них надати початкове значення (на власний розсуд). Значення полів другої структурної змінної ввести з клавіатури. Програма повинна порівнювати структурні змінні `P` та `Q` та друкувати на екран повідомлення, чи співпадають вони.
4. Для чого слугує в мові С засіб **typedef**, як він застосовується?
5. Переписати програми з попередніх завдань, запровадивши для структурних типів скорочені імена.

[1] Використання префіксу «`tag`» у подібних конструкціях є неписаним правилом програмування мовою С.

9.3. Масиви структур

Ключові слова: масиви структур, програмна модель таблиць, звертання до поля структури-елементу масиву.

Робота з масивами структур відбувається за тими ж загальними правилами, що були викладені в розділі 6. Нижче показано два приклади оголошення масивів структур: за допомогою повного імені та із застосуванням **typedef**

```
#define N 20
struct tPoint {
  double x;
  double y;
};
struct tPoint chain[N];

typedef struct tagPerson {
  char name[20];
  char surname[20];
  int yearOfBirth;
} tPerson;
tPerson students[ N ];
```

Масиви структур особливо часто використовуються на практиці там, де потрібно обробляти таблиці. Справді, кожна структура — елемент масиву відповідає одному рядку таблиці; всі елементи масиву однотипні, складаються з однакових полів, що відповідають графам таблиці. Наприклад, оголошені вище масиви є програмними моделями таких таблиць:

x	y
...	...
...	...

Ім'я	Прізвище	Рік народж.
...	...	...
...	...	...

Покажемо на прикладах, як обробляються такі таблиці. Нехай таблицю `chain` потрібно заповнити координатами точок кола одиничного радіусу. Зрозуміло, що колу належать точки (x, y) при $x = \cos\alpha$, $y = \sin\alpha$. Тоді нам потрібно перебрати значення α від 0 до 2π з постійним кроком $\Delta\alpha = \frac{2\pi}{N}$. Тоді маємо такий програмний текст:

```
1  #include <stdio.h>
2  #include <math.h> /* для тригоном. ф-цій */
3  #define N 20
4  struct tPoint {
5    double x;
6    double y;
7  };
8  int main() {
9  struct tPoint chain[ N ];
10 int i;
11 double dalpha; /* крок по альфа */
12 dalpha = 2 * M_PI / N;
13 for( i = 0; i < N; i++ ) {
14   chain[i].x = cos( i * dalpha );
15   chain[i].y = sin( i * dalpha );
16 }
17 /* координати точок обчислено та
18 занесено в таблицю, тепер друкувати */
19 printf( "X\tY\n" );
20 for( i = 0; i < N; i++ )
21   printf( "%6lf\t%6lf\n",
22     chain[i].x, chain[i].y );
23 /* ще якісь обчислення ...*/
```

```
24 return 0;
25 }
```

Нагадаємо, що M_PI — це константа, визначена в заголовочному файлі math.h, значенням якої є число π з точністю, на яку здатний тип **double**.

Як видно з тексту програми, для звертання до поля структури, яка є елементом масиву, потрібно спочатку до масиву застосувати операцію індексування, тобто вибрати з масиву певний елемент (в даному прикладі chain[i]), а потім до отриманої структури застосувати операцію звертання до поля.

Покажемо приклад обробки масиву структур типу tPerson. Нехай користувач вводить з клавіатури особові дані для N осіб, а програма зберігає ці дані в масиві.

```
1  #include <stdio.h>
2  #define N 20
3  typedef struct tagPerson {
4    char name[20];
5    char surname[20];
6    int yearOfBirth;
7  } tPerson;
8  int main() {
9  tPerson students[ N ];
10 for( i = 0; i < N; i++ ) {
11   printf( "Введіть␣дані␣%d-ї␣особи\n", i+1 );
12   printf( "Ім'я:␣" );
13   scanf( "%s", students[i].name );
14   printf( "Прізвище:␣" );
15   scanf( "%s", students[i].surname );
16   printf( "Рік␣народження:␣" );
17   scanf( "%d", &(students[i].yearOfBirth) );
18 }
19 /* подальша обробка */
20 return 0;
21 }
```

Слід звернути увагу на рядок, де вводиться рік народження. Він ілюструє, що за допомогою операції взяття адреси & можна дізнатися адресу не лише «звичайної» змінної, але й поля структури. Справді, структура займає певне місце в пам'яті, і кожне її поле повинно десь розташовуватися. З тексту програми видно, що спочатку треба звернутися до поля структури (вираз students[i].yearOfBirth в дужках), а потім до цього виразу застосувати операцію &.

Запитання

1. Продовжити другий приклад: в масиві students знайти всіх осіб на ім'я Святослав і по кожній з них надрукувати всі дані (прізвище, ім'я, рік народження).
2. Продовжити другий приклад: знайти найстаршу та наймолодшу особу та надрукувати всі їх дані.

9.4. Покажчики на структури

Ключові слова: покажчики на структури, операція непрямого звертання до поля структури (звертання до поля через покажчик), динамічне виділення пам'яті під масиви структур.

Змінна типу покажчика на об'єкт структурного типу (для скорочення говорять просто «*покажчик на структуру*») оголошується за тими ж правилами, що були викладені в розділі 7. Розглянемо приклад (оголошення структурних типів tPoint та tPerson див. вище):

```
1 struct tPoint A = {1, 1}, B = {0, -1}, *p;
2 tPerson theLeader, theProgrammer, *q;
3 p = &A;
4 q = &theProgrammer;
```

Рядки, помічені цифрами 1 та 2, ілюструють, що можна поєднувати оголошення змінних структурного типа з оголошеннями змінних типу покажчиків на структури. Різниця між цими рядками в тому, що в першому використовується повне ім'я структурного типу, а в другому — скорочене. Оператори, відмічені цифрами 3 та 4, слугують прикладами того, як операція взяття покажчика застосовується до змінних структурного типу.

Розберемо тепер, як працювати зі структурним об'єктом, на який вказує покажчик. Нехай потрібно виконати таку дію: в об'єкті, адреса якого міститься у покажчику `p`, взяти поле `x` та присвоїти йому значення -5. Уважний читач вже знає з попередніх розділів всі потрібні для цього засоби: спочатку розіменуємо покажчик `p`, тобто від покажчика перейдемо до об'єкту, на який він вказує; потім до полів цього об'єкту можна звертатися за загальним правилом, через операцію «крапка». В підсумку маємо:

```
(*p).x = -5;
```

Розглянута конструкція «*взяти об'єкт, на який вказує покажчик, та звернутися до поля цього об'єкту*» настільки часто буває потрібна на практиці, що для неї в мові С запроваджено спеціальне, зручніше позначення:

```
p->x = -5;
```

Цей оператор нічим не відрізняється від попереднього, лише пишеться коротше. Операцію `->` називають *непрямим звертанням до поля структури*.

Корисно буде розглянути такий фрагмент програми:

```
struct tPerson theStudent, *w;
w = &theStudent;
strcpy( w->name, "Володимир" );
```

Тут операція непрямого звертання використовується для того, щоб отримати адресу символьного масиву (текстового рядка).

Розглянемо насамкінець особливості динамічного виділення пам'яті для структур. Нижче дано приклад програми, яка запитує у користувача кількість осіб, динамічно створює масив структур типу `tPerson` в точності потрібного розміру, вводить та заносить до масиву дані, потім якось обробляє їх та нарешті звільняє пам'ять.

```
1 #include <stdio.h>
2 #include <alloc.h>
3 #include <stdlib.h>
4 typedef struct tagPerson {
5   char name[20];
6   char surname[20];
7   int yearOfBirth;
8 } tPerson;
9 int main() {
10 tPerson *p;
11 int n, i;
12 printf( "Введіть␣кількість␣" );
13 scanf( "%d", &n );
14 p = (tPerson*) malloc(n * sizeof(tPerson));
15 if( p == NULL ) {
16   printf( "немає␣пам'яті\n" );
17   return -1;
18 }
19 for( i = 0; i < n; i++ ) {
20   printf("Введіть␣дані␣%d-ї␣особи\n", i+1);
```

```
21    printf( "Ім'я: " );
22    scanf( "%s", p[i].name );
23    printf( "Прізвище: " );
24    scanf( "%s", p[i].surname );
25    printf( "Рік народження: " );
26    scanf( "%d", &(p[i].yearOfBirth) );
27  }
28  /* подальша обробка */
29  free( p );
30  return 0;
31  }
```

Як видно з прикладу, динамічний розподіл пам'яті для масивів структур робиться за тими ж загальними правилами, що були описані в розділі 7.5. А саме, для роботи з масивом оголошується змінна типу покажчика. Оскільки в даній задачі масив складається зі структур, то й змінна `p` має тип покажчика на структуру. Виділення пам'яті здійснює функція `malloc`. Розмір області пам'яті, який передається їй в якості аргументу, обчислюється як добуток розміру одного елементу (операція **`sizeof`**) на кількість елементів (змінна `n`).

Далі робиться перевірка, чи успішно виділено пам'ять. До елементів динамічно створеного масиву можна звертатися, як звичайно, за допомогою операції індексування. Наприкінці програми потрібно звільнити динамічно виділену область пам'яті, викликавши функцію `free`.

Запитання

1. Що означає операція `->`, як можна замінити її іншими операціями?
2. Продовжити приклад зі с. 121: нехай програма запитує у користувача число N та динамічно виділяє пам'ять для масиву точок довжини N.

9.5. Структури і функції

Ключові слова: аргументи структурного типу, функції, що повертають значення структурного типу.

Функції в мові C можуть мати аргументи структурного типу та можуть повертати значення структурного типу. Згадавши оголошений в попередніх прикладах тип `tPoint`, що моделює точку в декартовій системі координат, напишемо функцію, яка обчислює відстань між двома точками.

Функція, очевидно, повинна мати два аргументи типу «точка» та повертати дійсне число, відстань. Дамо функції ім'я `distance`. Для наочності наведемо повний текст програми, яка містить оголошення та означення цієї функції, а також функцію `main`, яка запитує у користувача координати точки Q та за допомогою функції `distance` обчислює відстань між нею та заданою точкою $M(12, 8)$.

```
1  #include <stdio.h>
2  #include <math.h>
3
4  typedef struct tagPoint{
5    double x;
6    double y;
7  } tPoint;
8
9  double distance(tPoint a, tPoint b);
10
11 int main() {
12 tPoint M = { 12, 8 }, Q;
13 printf( "Введіть координати точки Q " );
14 scanf( "%d %d", &(Q.x), &(Q.y) );
```

```
15 printf("Відстань␣між␣точками␣M␣та␣Q␣складає␣");
16 printf( "%lf\n", distance(M,Q) );
17 return 0;
18 }
19
20 double distance(tPoint a, tPoint b) {
21 double dx = a.x - b.x;
22 double dy = a.y - b.y;
23 double s = dx*dx + dy*dy;
24 return sqrt( s );
25 }
```

Тут змінні `M` та `Q` є локальними змінними функції `main`. В момент виклику функції `distance`, як і було описано в розділі 5.4, значення цих змінних присвоюються змінним `a` та `b`, що є аргументами функції `distance`. Єдиною особливістю є те, що ці значення є структурами, отже копіюються вони поле за полем. Після цього управління передається на тіло функції `distance`.

Внутрішня будова функції `distance` достатньо очевидна. Локальні змінні `dx` та `dy` є, відповідно, різницями x- та y-координат точок-аргументів, змінна `s` — сумою квадратів цих різниць. Тоді відстанню між точками, а отже і значенням, яке має повернути функція, є квадратний корінь з `s`.

Тепер розглянемо функцію, яка повертає значення структурного типу. Нехай функція `polar2carth` приймає два аргументи дійсного типу — координати точки в полярній системі координат (відстань та кут), переводить їх у декартову систему і повертає результат у вигляді структури типу `tPoint`.

```
1 tPoint polar2carth( double r, double phi );
2
3 int main() {
4 double u, v;
5 tPoint A;
6 printf( "Введіть␣полярні␣координати␣" );
7 scanf( "%lf␣%lf", &u, &v );
8 A = polar2carth( u, v );
9 printf( "B␣декартовій␣системі:␣" );
10 printf( "(%lf,␣%lf)\n", A.x, A.y );
11 return 0;
12 }
13
14 tPoint polar2carth( double r, double phi ) {
15 tPoint M;
16 M.x = r * cos( phi );
17 M.y = r * sin( phi );
18 return M;
19 }
```

При виклику функції `polar2carth` створюється локальна змінна `M` структурного типу `tPoint`. Її полям `x` та `y` присвоюються декартові координати точки, обчислені за загальновідомими формулами. Коли функція `polar2carth` повертає структурне значення до функції `main`, воно присвоюєься у змінну `A`. Тобто поля структурної змінної `M` з функції `polar2carth` копіюються у поля змінної `A` з функції `main`.

1. Написати функцію, яка приймає в якості аргумента об'єкт типу `tPerson` (особові дані про людину) та повертає вік людини в роках (ціле число).
2. Написати функцію `middle`, яка приймає два аргументи типу **struct** `tPoint` — точки A і B та повертає об'єкт цього ж типу — точку M, що є серединою відрізку AB.
3. Написати функцію з іменем `nearest`, яка приймає аргумент типу **struct** `tPoint` — координати деякої точки M, покажчик на масив точок та ціле число — довжину цього масиву, та повертає ту точку з масиву, яка лежить найближче до точки M.

9.6. Підсумковий огляд

Цей розділ присвячено більш досконалим засобам роботи з даними, які надає мова C. Як видно з попередніх розділів, базовий, найпростіший рівень організації даних становлять числа (цілі, дійсні чи коди літер). Їх можна назвати простими, або атомарними в тому розумінні, що такі значення не складаються з більш простих частин. Вище також розглядалися масиви. Це складні, структуровані дані: кожен масив складається з багатьох елементів, причому всі елементи в масиві мають однаковий тип і кожен елемент має свій номер.

Структури, які також називають записами, це такі складні дані, які містять в собі дані-елементи різних типів, а елементи розрізняються між собою за іменами.

З точки зору мови C структури являють собою тип даних, створений самим програмістом. Іншими словами, мова C надає програмісту набір елементарних типів даних (**int**, **double**, **char**, деякі інші) та засоби, за допомогою яких можна з цих «цеглинок» побудувати власні типи та надбудувати ними мову. Такими засобами є масиви, покажчики та структури.

Вище вже згадувалося, що апарат складних даних в мові C логічно повний: складні типи даних можна конструювати не лише з елементарних даних, але й з інших, сконструйованих раніше, складних типів. Наприклад, полями структур можуть бути масиви, можна обробляти масиви структур, можна оголошувати структури, полями яких є структури і масиви структур і т.д. до нескінченності.

Розділ 10

Обробка файлів, частина 1

10.1. Основні поняття

Ключові слова: файл, відкриття файлу, закриття файлу, файлове введення та виведення, текстовий файл.

Програми, які розглядалися в попередніх розділах, брали дані лише з клавіатури та виводили результати своєї роботи лише на екран. Звичайно ж, програма повинна вміти записувати дані для довготривалого зберігання на диск та зчитувати їх. В абсолютній більшості сучасних операційних систем дані на носії (здебільшого, диску) упорядковуються у *файли*. Файл — це іменована сукупність даних, що зберігається на пристрої.

Мова C містить надзвичайно вдало продуману систему засобів для обробки файлів. Мова підтримує два основних способи роботи з файлами — на низькому рівні через дескриптори та на високому рівні через потоки. В цьому ознайомчому курсі будемо розглядати лише другий спосіб (перший вивчатиметься в курсі системного програмування).

Будь-який файл являє собою послідовність байтів. В залежності від того, який сенс та призначення мають ці байти, файли поділяють на текстові та двійкові. Текстові файли можуть містити лише ті байти, що є кодами алфавітних, цифрових символів, знаків пунктуації, пробілів, табуляцій та символи переходу на новий рядок. Двійкові файли можуть містити будь-які значення байтів, в тому числі і такі, яким не відповідає графічний знак, що може бути виведений на екран. В цій главі розглядаємо лише текстові файли, обробці двійкових файлів присвячено наступну главу.

З точки зору програміста робота з файлами виглядає так. Програма безпосередньо оперує не з голівкою магнітного диску, а з допоміжною змінною (скажімо, `f`) типу покажчика на певні службові дані, через які вже операційна система отримує доступ до конкретного місця на диску. Перш ніж робити з файлом на диску будь-які дії (читати дані з файлу чи писати у файл), програма повинна його *відкрити* — при цьому файл, розташований на диску, *зв'язується* зі змінною `f` (тобто до змінної заносяться службові дані для доступу саме до даного файлу).

Після цього, коли треба записати чи прочитати дані з файлу, програміст викликає спеціальні функції читання та запису, передаючи їм через один аргумент змінну `f` — канал доступу до файлу, а через решту аргументів — які саме дані читати чи писати. Насамкінець програма повинна закрити файл — розірвати зв'язок між змінною `f` та файлом на диску, при цьому звільняються ресурси операційної системи, що використовуються для доступу до файла.

Для початку розберемо приклад програми, яка створює на диску файл під назвою «`greeting.txt`» та записує в нього текстові повідомлення і числа.

```
1  #include <stdio.h>
2  int main() {
3    FILE *f;
4    char theName[] = "greeting.txt";
5    int theYear = 884;
6    f = fopen( theName, "w" );
7    fprintf(f, "Древляни,␣рік␣%d␣і␣до␣наших␣днів\n",
8        theYear );
9    fclose( f );
10   return 0;
11 }
```

Основні функції файлового введення-виведення оголошено в тому ж заголовочному файлі

`stdio.h`, що й функції введення з клавіатури та виведення на екран[1].

Вся подальша робота з файлом здійснюється через змінну `f` — покажчик на структуру даних типу `FILE`. Які конкретно дані в ній зберігаються, програмісту немає потреби знати.

Увага! Дуже розповсюджена груба помилка початківців — вважати, що «`f` — це файл» (або, ще гірше, «`f` — це покажчик на файл»: покажчик може бути лише на об'єкт даних, що зберігається в пам'яті). Насправді `f` — це просто покажчик на допоміжну структуру даних, через посередництво якої програма звертається до справжнього файлу, який розташовано на диску. Ця структура даних має тип під назвою «`FILE`», але ж сама по собі вона не є файлом.

Далі в програмі оголошено текстовий рядок `theName` з вмістом «`greeting.txt`». Цей текст в подальшому буде використано в ролі імені файлу (це те ім'я, під яким файл буде зберігатися на диску, зокрема, за цим ім'ям його зможуть «побачити» інші програми). Звертаємо увагу: в цьому рядку не створюється файл з таким ім'ям, а лише оголошується текстовий рядок.

Змінна `theYear`, значенням якої є рік заснування Житомира, потрібна в цьому прикладі для того, аби продемонструвати запис у файл числа в десятковій системі.

Далі відбувається відкриття файлу. Функція `fopen` має два аргументи. Першим аргументом є покажчик на текстовий рядок, що містить ім'я файлу, який треба відкрити, а другий аргумент — режим відкриття (як саме відкрити файл). Текст «`w`» в другому аргументі означає «відкрити файл для запису; якщо файлу з таким ім'ям ще немає на диску, створити порожній файл, а якщо вже є — стерти з нього весь попередній вміст». Отже, після цього виклику на диску в обох випадках з'явиться порожній файл під іменем «`greeting.txt`», готовий для того, щоб записувати в нього інформацію.

Значення, яке повертає функція `fopen` — покажчик на службову структуру даних, через яку програма може звертатися до файлу на диску, присвоюється у змінну `f`. Відтепер ця змінна є посередником між програмою та дисковим пристроєм.

В наступному рядку викликається функція `fprintf`, яка виводить у файл звичайний текст та відформатовані у текстовому вигляді числа. Читач легко помітить, що ця функція дуже схожа на функцію виведення на екран `printf`. Єдина відмінність полягає в тому, що перед форматним рядком з'явився ще один аргумент — покажчик на службову структуру даних, через яку відбувається доступ до файлу. Функція `fprintf` виводить текст так само, як і функція `printf`, але не на екран, а у файл.

Далі викликається функція `fclose`, яка закриває файл, пов'язаний зі змінною `f`, тобто розриває зв'язок між службовою структурою даних в пам'яті машини та файлом на диску.

Увага! Не треба забувати закривати файл одразу, як закінчилася робота з ним. Цим програма економить системні ресурси та зменшує ймовірність збоїв.

Після запуску описаної програми на диску повинен виникнути текстовий файл з іменем «`greeting.txt`», в якому міститься текст «Древляни, рік 884 і до наших днів» з символом переходу на новий рядок наприкінці.

Запитання

1. Що таке файл? В чому відмінність між текстовими та двійковими файлами?
2. Яку роль в програмі відіграє змінна `FILE *f`?
3. Які функції відкривають та закривають файли?

10.2. Відкриття та закриття файлу

Ключові слова: відкриття файлу, закриття файлу, потік, режим відкриття, помилка при відкритті файлу, `fopen`, `fclose`.

Ще раз нагадаємо, що `FILE` — це ім'я структурного типу даних, означення якого міститься в заголовочному файлі `stdio.h`. Надалі будемо називати об'єкти цього типу *потоками*,

[1] Це не випадково: одна з важливих засад мови C полягає в тому, щоб розглядати екран, клавіатуру та інші пристрої як особливі файли — див. розділ 10.7.

Табл. 10.1. Режими відкриття файлів у функції `fopen`

`r`	Лише читання	з початку	Файл мусить існувати
`r+`	Читання і запис	з початку	Файл мусить існувати
`w`	Лише запис	з початку	Якщо не існує, створюється
`w+`	Читання і запис	з початку	Якщо не існує, створюється
`a`	Лише запис	в кінець	Якщо не існує, створюється
`a+`	Читання і запис	в кінець	Якщо не існує, створюється

також будемо вживати для таких покажчиків, як змінна `f` з попереднього прикладу, термін «*покажчик на потік*».

Для відкриття файлу слугує функція `fopen`, прототип якої у файлі `stdio.h` має вигляд:

```
FILE* fopen( char *filename, char *mode );
```

Це означає, що функція має два аргументи, обидва типу покажчиків на текстові рядки, а повертає значення типу покажчика на потік. Перший аргумент функції — ім'я файлу, який треба відкрити. Це може бути або просто ім'я (наприклад, «`greeting.txt`») — в такому випадку по замовчуванню місцем розташування файлу система вважає поточний каталог, або ім'я разом зі шляхом по підкаталогах. Треба мати на увазі, що в операційних системах DOS та Windows символом, що розділяє імена підкаталогів, є обернена похила риска \. В мові C за цим символом зарезервовано використання у складі спеціальних комбінацій символів (`\n`, `\t` та ін.), тому сам символ \ в текстових константах позначається так: \\. Наприклад, щоб відкрити файл `aaa.dat`, розташований у підкаталозі `data` каталогу `stars` на диску `D`, треба ім'я файлу задати в такому вигляді:

```
f = fopen( "D:\\stars\\data\\aaa.dat", "r" );
```

Другий аргумент функції позначає режим відкриття файлу: від нього залежить, які операції з файлом в подальшому можуть виконуватися. Режими показано в табл. 10.1.

Відкриття файлу може відбутися з успіхом або з неуспіхом. У випадку успіху функція `fopen` створює в пам'яті службову структуру даних — потік, зв'язаний з потрібним файлом на диску, та повертає покажчик на цей потік. Цей покажчик потрібно зберегти в деякій змінній та потім використовувати в усіх операціях введення-виведення (див. приклад вище).

Неуспіх при спробі відкрити файл може виникнути з таких причин: для імені файлу, якого не існує на диску, задано режим `r` або `r+`; програма намагається відкрити для запису файл, для якого операційна система дає доступ лише для читання; програма відкрила надто багато файлів і вичерпала ліміт дозволених ресурсів тощо. В усіх таких випадках, коли замовлений файл в замовленому режимі відкрити неможливо, функція `fopen` повертає значення `NULL`.

Поведінка програми при неуспішному відкритті файлу та ще в багатьох схожих ситуаціях заслуговує на окремий розгляд. Гарно написана програма повинна бути стійкою до помилок, що можуть виникати під час виконання, через обставини, які програмісту та його програмі непідконтрольні. Скажімо, якщо програма запитує ім'я файлу у користувача, цілком може статися, що користувач помилково введе ім'я файлу, який насправді не існує. Або програма намагається записати дані у файл, а наявні у користувача права доступу забороняють запис.

Звичайно ж, програма повинна розпізнати таку нештатну ситуацію та коректно обробити її. Наведемо кілька типових рішень. В найпростішому випадку програма, побачивши, що файл відкрити неможливо, просто завершує свою роботу, друкуючи на екран повідомлення.

```
1 #include <stdio.h>
2 int main() {
3   char fileName[ 80 ];
4   FILE *f;
5   printf( "Введіть␣ім'я␣файлу:␣" );
6   scanf( "%s", fileName );
7   f = fopen( fileName, "r" );
```

```
8    if( f == NULL ) { /* перевірити, чи відкрився */
9      printf( "%s␣неможливо␣відкрити\n", fileName );
10     return -1; /* завершити програму */
11   }
12   /* далі нормальна обробка файлу */
13   . . .
14     fclose( f );
15   return 0;
16 }
```

Нагадаємо, що функція main повинна повертати ціле число, причому 0 символізує, що програма відпрацювала успішно, а будь-яке інше число означає, що програма завершилася через помилку.

Наведений вище фрагмент працює за доволі примітивною логікою, завершуючи програму при невдачі. Бажано зробити програму більш гнучкою: у випадку невдачі вона пропонує користувачу ввести інше ім'я файлу або, якщо користувач бажає, завершити роботу.

```
1 #include <stdio.h>
2 int main() {
3   char fileName[ 80 ];
4   FILE *f;
5   do {
6     printf("Введіть␣ім'я␣файлу␣або␣крапку:␣" );
7     scanf( "%s", fileName );
8     if( strcmp( fileName, "." ) == 0 )
9       return 0;
10    f = fopen( fileName, "r" );
11  } while( f == NULL );
12  /* далі нормальна обробка файлу */
13  . . .
14  fclose( f );
15  return 0;
16 }
```

Проаналізуємо логіку роботи цієї програми. Цикл з постумовою спочатку виконує тіло, а потім перевіряє, чи продовжувати процес. Тому при першому вході в цикл програма запитає у користувача ім'я файлу. Якщо користувач вводить ім'я неіснуючого файлу, функція fopen повертає значення NULL, яке присвоюється в змінну f. Тоді при перевірці умови продовження циклу порівняння f==NULL дасть значення «істина», і оператор циклу буде повторювати описану процедуру ще раз.

Якщо ж користувач замість імені файлу введе крапку, функція main поверне значення 0, а вихід з головної функції означає завершення роботи програми.

Якщо виконання програми дійшло до наступного після циклу оператору, то це могло відбутися тільки тоді, коли після черговової ітерації умова продовження (f==NULL) виявилася хибною, а це, в свою чергу, можливо лише тоді, коли функції fopen вдалося відкрити файл з вказаним іменем.

Запитання

1. Які аргументи приймає та яке значення повертає функція fopen?
2. Як відкрити файл для читання; для дописування в кінець; для читання та запису; для перезапису, тобто зі знищенням його попереднього вмісту?
3. Як після спроби відкрити файл перевірити, чи вдалося його відкрити?

10.3. Введення-виведення тексту

Ключові слова: функції читання та запису, fprintf, fputs, fscanf, fgets, кінець файлу,

`feof`.

Якщо текстовий файл відкрито в режимі, що допускає читання, то прочитати з нього довільні дані можна за допомогою функції `fscanf`. Це майже повний аналог функції `scanf`. Перший аргумент — покажчик на потік, другий аргумент — форматний рядок, що містить будь-яку кількість специфікаторів формату, решта аргументів (їх кількість відповідає кількості специфікаторів) — покажчики на змінні, в які треба розмістити результати розбору прочитаного тексту.

Функція `fscanf` намагається читати символи з потоку та співставляти їх з форматним рядком так само, як функція `scanf` розбирає послідовність символів, введену з клавіатури. Функція повертає число успішно співставлених специфікаторів, детальніше див. розділ 2.8.

Нехай, наприклад, дано файл з іменем «`data.txt`» з таким вмістом:

```
15 -37.511
князь Святослав Ігоревич
```

Розглянемо програму (щоб не засмічувати розгляд, в тексті програми пропущено перевірку на помилку відкриття)

```
1  #include <stdio.h>
2  #define LEN 256
3  int main() {
4    FILE *f;
5    int m, n;
6    double dt;
7    char s[ LEN ];
8    f = fopen("data.txt", "r");
9    n = fscanf("%d␣%lf␣%s", &m, &dt, s);
10   printf("Прочитано␣%d␣значень:\n", n);
11   printf("Ціле␣%d,␣дійсне␣%lf,␣рядок␣%s\n",
12       m, dt, s);
13   fclose( f );
14   return 0;
15 }
```

Ця програма успішно відкриє файл для читання. Функція `fscanf` прочитає з файлу символи «`1`» та «`5`» і, співставивши їх зі специфікатором «`%d`», перетворить їх на ціле число 15 та запише це число в змінну `m`. Точно таким же чином функція прочитає наступні знаки та занесе в змінну `dt` значення $-37,511$. Далі, обробляючи специфікатор «`%s`», функція буде намагатися знайти у файлі послідовність символів до першого роздільника (пробіла, табуляції або переходу на новий рядок). Таким чином, функція `scanf` прочитає слово «князь», бо за ним йде пробіл, занесе його у масив `s`, приписавши до кінця нуль-символ кінця рядку. Решту тексту з файлу програма не прочитає взагалі (якби програма далі вводила ще дані цього файлу, вона прочитала б залишок, починаючи зі слова «Святослав»).

Оскільки функція `fscanf` успішно співставила всі три специфікатори, вона поверне значення 3, яке і буде присвоєно змінній `n`. Тому програма надрукує на екран такий результат:

```
Прочитано 3 значень:
Ціле 15, дійсне -37.511, рядок князь
```

Для введення текстового рядка, що містить пробіли та табуляції, слугує спеціальна функція `fgets`. Вона має такий прототип:

```
char *fgets( char *s, int n, FILE *f );
```

З прототипу видно, що функція має три аргументи: покажчик на символьний масив, в якому буде розміщено прочитаний з файлу текстовий рядок, ціле число — границю довжини рядка, та покажчик на потік — файл, з якого треба читати рядок. Функція повертає покажчик на ту ж саму область пам'яті `s`, в яку розміщено прочитаний рядок. Функція читає текст з файлу, літера за літерою, поки не зустріне символ кінця рядка, кінець файлу, або поки не прочитає `n-1` символ.

Для прикладу розглянемо той же файл «`data.txt`» та трохи видозмінену програму (для скорочення наведемо лише основну частину):

```
fscanf("%d␣%lf\n", &m, &dt);
fgets(s, LEN, f);
printf("Ціле␣%d,␣дійсне␣%lf,␣рядок␣%s\n",
  m, dt, s);
```

Спочатку функція `fscanf` прочитає числа 15 та $-37,511$ разом з символом переходу на новий рядок. Таким чином, наступна операція введення буде читати решту файлу починаючи з символу «к». Отже, функція `fgets` прочитає весь рядок «князь Святослав Ігоревич» та занесе його в масив `s`.

Функція `fgets` безпечна в тому сенсі, що, завдяки параметру `n`, не дозволить ввести символів більше, ніж вміщається в зарезервовану область пам'яті. Ще раз змінимо розглянуту вище програму, зменшивши довжину символьного масиву (константа `LEN`). Текст зміненої програми наводимо повністю:

```
1  #include <stdio.h>
2  #define LEN 10
3  int main() {
4    FILE *f;
5    int m, n;
6    double dt;
7    char s[ LEN ];
8    f = fopen("data.txt", "r");
9    fscanf("%d␣%lf\n", &m, &dt);
10   fgets(s, LEN, f);
11   printf("Ціле␣%d,␣дійсне␣%lf,␣рядок␣%s\n",
12       m, dt, s);
13   fclose( f );
14   return 0;
15 }
```

Буфер `s` для розміщення тексту тепер може вмістити не більше 9 символів (бо десятий потрібен для завершального нуль-символу). Рядок «князь Святослав Ігоревич» значно довший. Але функція `fgets`, «знаючи» завдяки другому аргументу про обмеження довжини, прочитає з файлу та розмістить в масив `s` лише перші 9 літер «князь Свя», допише в десятий елемент нуль-символ, а решта символів у файлі залишаться непрочитаними (їх могли б прочитати наступні операції введення, якби вони були в програмі).

Існує набір функцій, які записують інформацію у файл. Звичайно ж, для цього файл має бути відкритий в режимі, що допускає запис. Функція `fprintf` виводить у файл відформатовані дані. Її перший аргумент — покажчик на потік, другий — форматний рядок, решта аргументів — значення довільних типів, що відповідають специфікаторам. Працює повністю подібно до функції `printf`, але виводить результат не на екран, а у файл.

Функція `fputs` призначена спеціально для виведення у файл текстового рядка. Має прототип

```
int fputs( char *s, FILE *f );
```

Перший аргумент — покажчик на текстовий рядок, який треба записати, а другий — покажчик на потік, до якого треба записати рядок.

Обробка файлу найчастіше полягає в тому, щоб читати з нього дані до тих пір, поки файл не закінчився. Ключове питання тут — як дізнатися, що файл вже закінчився. Це робиться за допомогою функції

```
int feof( FILE *f );
```

Ця функція приймає аргумент — покажчик на потік та повертає значення 0, якщо файл ще не закінчився, і значення 1, якщо в процесі читання досягнуто кінець файлу. Приклад застосування буде показано в наступному розділі.

Запитання

1. Яка функція дозволяє читати з текстового файлу довільні дані (числові цілі та дійсні, текст тощо)?
2. Яка функція дозволяє читати з текстового файлу текстові рядки з контролем довжини?
3. Яка функція дозволяє виводити текстовий рядок до файлу?
4. За допомогою якої функції можна перевірити, чи не досягнуто в процесі читання кінець файлу?
5. Написати програму, яка читає з текстового файлу 10 чисел та в інший файл записує їх квадрати.

10.4. Простий приклад

Ключові слова: цикл читання до закінчення файлу, коректна обробка помилок у вхідному файлі.

Розберемо приклад програми, який ілюструє типові прийоми обробки текстових файлів. Нехай дано текстовий файл, в якому вміщено цілі числа (записані цифрами в десятковій системі). Напишемо програму, яка обчислює суму цих чисел та підраховує кількість нулів серед чисел. Результати роботи програма записує в інший текстовий файл.

```
1  #include <stdio.h>
2  #define LEN 80
3  int main() {
4    int x, sum=0, n0=0, k;
5    FILE *in, *out;
6    char fileName[ LEN ];
7    printf( "Ім'я␣файлу␣даних?␣");
8    scanf("%s", fileName);
9    in = fopen(fileName, "r");
10   if( f == NULL ) {
11     printf("Неможливо␣відкрити␣для␣читання\n");
12     return -1;
13   }
14   printf("Ім'я␣файлу␣результатів?␣");
15   scanf("%s", fileName);
16   out = fopen(fileName, "w");
17   if( out == NULL ) {
18     printf("Неможливо␣відкрити␣для␣запису\n");
19     fclose(in);
20     return -1;
21   }
22   while( !feof(in) ) {
23     k = fscanf( in, "%d", &x );
24     if( k != 1 ) {
25       fprintf(out, "У␣вхідному␣файлі␣не␣число\n");
26       break;
27     }
28     s += x;
29     if( x == 0 )
30       ++ n0;
31   }
32   fprintf( out, "Сума␣%d,␣нулів␣%d\n", sum, n0 );
33   fclose(in);
34   fclose(out);
35   return 0;
36 }
```

Ця програма має обробляти одночасно два файли. Тому в ній оголошено дві змінні типу покажчика на потоки: змінна `in` відповідає вхідному потоку, а змінна `out` — потоку для виведення результатів. Після спроби відкрити файл з даними робиться звичайна перевірка, чи успішно цей файл відкрився, у разі неуспіху програма завершується. Якщо ж виконання програми дійшло до рядка 14 то це значить, що вхідний файл відкрився нормально.

Ім'я файлу використовується лише один раз, при відкритті. Після того, як файл відкрито, програма «спілкується» з ним лише через потік, пам'ятати ім'я файлу програмі більше не потрібно. Тому ім'я другого файлу розміщується у тому ж масиві `fileName`, затираючи попереднє ім'я вхідного файлу.

У випадку, коли не вдається відкрити файл для зберігання результатів, програма завершується, але перед цим вона мусить закрити потік `in`, оскільки його вже відкрито.

Далі починається основна частина програми. Логіка її роботи полягає в тому, щоб читати з файлу числа по одному, поки не буде досягнуто кінець файлу. Для цього використовується цикл (рядок 22). Цикл повторюється доти, доки функція `feof` не поверне значення 1 (звертаємо увагу на операцію заперечення), а таке значення вона поверне тільки тоді, коли файловий (потік) `in` буде прочитано до кінця.

Прочитавши чергове число з файлу у змінну `x`, програма додає його до суми та, якщо воно дорівнює 0, збільшує лічильник нулів.

Після завершення циклу, коли весь файл оброблено, програма виводить до потоку `out` звіт про результати.

Особливу увагу треба звернути на рядок 24. Він захищає програму від нештатної ситуації, коли у вхідному файлі є недопустимі дані — будь-який текст, крім чисел. Якщо у вхідному файлі виявиться будь-який зайвий символ (скажімо, літера «a»), то функція `scanf` не зможе співставити цей символ зі специфікатором «`%d`» та поверне значення 0 (бо оброблено 0 специфікаторів). В такому випадку програма записує у файл результатів повідомлення про помилку та припиняє спроби читати файл далі — цикл читання чисел розривається, наступні після циклу оператори виводять у файл результатів звіт про обробку тієї частини вхідного файлу, яку програма встигла обробити.

Цей маленький фрагмент програми насправді ілюструє важливий для професійного програмування прийом — програма не повинна покладатися на користувача, вона має бути стійкою до людських помилок, особливо у файлах, що передаються їй для обробки. Якщо файл, що подається на вхід програми, готує людина (наприклад, набирає вміст файлу у текстовому редакторі), то у файлі завжди можуть трапитися помилки, живі люди від них не застраховані. Тому добре написана програма повинна вміти, по-перше, розпізнати сам факт наявності помилки, по-друге, повідомити користувача про характер та місце розташування помилки, а по-третє, коректно продовжити або коректно завершити роботу.

Запитання

1. Написати програму, яка з одного файлу читає цілі числа, а в інший файл записує ті з них, що діляться на 3.
2. Написати програму, яка читає з файлу цілі числа та визначає, чи розташовані вони у файлі у порядку зростання, та друкує відповідне повідомлення.
3. Написати програму, яка читає з файлу цілі числа та знаходить і друкує на екран найбільше з них.

10.5. Посимвольна обробка

Ключові слова: `fputc`, `fgetc`.

Для деяких задач буває зручно вводити файл посимвольно — один символ за один раз. Для цього слугує функція `fgetc` з таким прототипом:

```
int fgetc( FILE *f );
```

Ця функція читає з файлу один символ та повертає його числовий код. Для запису у файл одного символу призначена функція з таким прототипом:

```
int fputc( int c, FILE *f );
```

Ця функція приймає два аргументи: код символу та покажчик на потік, та записує символ у потік.

Розглянемо типову задачу. Нехай у вхідному файлі міститься текст будь-якою людською мовою, причому в основний текст вставлено примітки, усілякі другорядні нотатки, взяті в кутові дужки «<» та «>». Програма повинна вилучити примітки з тексту, тобто перезаписати у другий файл весь текст з першого, пропускаючи при цьому всі символи, крім послідовностей, що починаються зі знаку «<» та закінчуються знаком «>».

Наприклад, у вхідному файлі міститься текст

```
Мова програмування C<Керніган та Річі> широко розповсюджена<майже все
системне програмне забезпечення для ОС UNIX, Windows>, забезпечує високу
ефективність<майже як при програмуванні мовою асемблера> та добре
стандартизована<стандарти ANSI та ISO>.
```

Тоді програма повинна у файл результатів записати такий текст:

```
Мова програмування C широко розповсюджена, забезпечує високу ефективність та
добре стандартизована.
```

Розв'язуючи цю задачу, заодно проілюструємо важливий принцип створення програм, на якому ґрунтується «велике» професійне програмування. Принцип полягає в тому, що жодна серйозна програма не створюється одразу ж в готовому вигляді. Натомість спочатку часто роблять ескіз майбутньої програми — початкову версію, завідомо дуже недосконалу, що аби як розв'язує поставлену задачу. В ескізній версії можна навіть спростити постановку задачі, зробити дещо не те, що хотів замовник.

Користь від такого грубого наближення полягає в тому, що програміст може на ньому набути досвід розв'язання нової для себе задачі (як позитивний, так і негативний, що теж цінно), намітити ідеї для побудови більш досконалих версій програми. Тим більше, що якщо система достатньо складна, то деякі її аспекти просто неможливо продумати та передбачити заздалегідь, їх можна дослідити лише експериментально — зробивши пробну версію програми. Далі розробка проходить низку проміжних стадій, кожна наступна версія досконаліша за попередню, бо в ній враховуються помилки та переваги попередніх версій.

В цьому параграфі зробимо першу версію програми, а в наступному параграфі значно вдосконалимо її.

В першому наближенні логіку роботи програми зрозуміти просто: вона читає символ за символом з вхідного файлу та одразу записує їх у файл-результат, поки не зустріне символ початку примітки «<». Побачивши цей символ, програма змінює свою поведінку: вона продовжує читати з вхідного файлу символ за символом, але тепер не копіює їх у файл-результат. Якщо програма, працюючи у такому режимі, «побачить» символ кінця примітки «>», то вона знов перемкнеться на перший режим роботи.

Опишемо цю логіку більш детально. Програма в кожен конкретний момент часу знаходиться або в режимі «текст», або в режимі «примітка». На кожному кроці вона читає з вхідного файлу один символ та обробляє його за такими правилами:

- Якщо поточним символом є «<», то перемикнутися в режим «примітка» та продовжити обробляти текст далі;
- Якщо поточним символом є «>», то перемикнутися в режим «текст» та продовжити обробляти текст далі;
- Якщо поточний символ не є одним з двох, що згадуються у попередніх пунктах, та якщо поточним режимом є «текст», то вивести цей символ у файл-результат та продовжити обробляти текст далі.

Описану логіку реалізує наведена нижче програма. В ній змінна `mode` означає поточний режим: значення 1 відповідає режиму «текст», а значення 0 — режиму «примітка». Зрозуміло, що програма починає свою роботу в стані «текст» — навіть якщо перший же символ файлу виявиться символом початку примітки, програма відповідно до правил перейде у стан «примітка».

```
1  #include <stdio.h>
2  #define LEN 80
3  int main() {
4    int x, mode=1;
5    FILE *in, *out;
6    char fileName[ LEN ];
7    printf( "Ім'я␣вхідного␣файлу?␣");
8    scanf("%s", fileName);
9    in = fopen(fileName, "r");
10   if( f == NULL ) {
11     printf("Неможливо␣відкрити␣для␣читання\n");
12     return -1;
13   }
14   printf("Ім'я␣файлу␣результатів?␣");
15   scanf("%s", fileName);
16   out = fopen(fileName, "w");
17   if( out == NULL ) {
18     printf("Неможливо␣відкрити␣для␣запису\n");
19     fclose(in);
20     return -1;
21   }
22   while( !feof(in) ) {
23     x = fgetc(in);
24     if( x == '<' ) { mode = 0; continue; }
25     if( x == '>' ) { mode = 1; continue; }
26     if( mode == 1 ) fputc(x, out);
27   }
28   fclose(in);
29   fclose(out);
30   return 0;
31 }
```

Запитання

1. Написати програму, яка читає текст з файлу та підраховує в ньому кількість пробілів.
2. Написати програму, яка читає текст з файлу та кожний другий його символ записує в інший файл.
3. Написати програму, яка читає текст з файлу та переписує його в інший файл, щоразу замінюючи літеру «о» на знак «*».
4. Написати програму, яка читає текст з файлу та знаходить довжину найбільшої послідовності знаків, що містяться між дужками «(» та «)». Вважати, що дужки у вхідному тексті не можуть бути вкладені: всередині однієї пари дужок не може міститися інша.

10.6. Посимвольна обробка: продовження

Алгоритм, покладений в основу попередньої програми, недосконалий. Це проявляється тоді, коли текст містить незбалансовані кутові дужки, наприклад:

```
перше >слово <друге <третє> > кінець <тексту
```

Програма повинна в процесі читання вхідного тексту перевіряти такі умови: чи не зайва закривальна кутова дужка (в прикладі перша закривальна дужка не має відповідної відкривальної)? Чи не відкривається примітка повторно? В прикладі одна приимітка починається перед словом «друге» а наступна — одразу ж після цього слова, до того, як закриється попередня примітка. Нарешті, чи закрито примітку до кінця файлу? В прикладі примітка починається перед словом «тексту», але на цьому слові файл закінчується, і отже в ньому немає відповідної закривальної дужки.

Щоб врахувати перераховані випадки, треба розробити детальніший перелік правил обробки символу. Тепер програма має обробляти кожний прочитаний символ за такими правилами:

- Якщо в режимі «текст» поточним символом є «<», то перейти в режим «примітка»;
- Якщо в режимі «текст» поточним символом є «>», то зафіксувати помилку — закривальна дужка без відповідної відкривальної;
- Якщо в режимі «текст» поточним символом є будь-який інший, то записати його до файлу-результату;
- Якщо в режимі «примітка» поточним символом є «<», зафіксувати помилку — повторне відкриття примітки;
- Якщо в режимі «примітка» поточним символом є «>», то встановити режим «текст»;
- Якщо в режимі «примітка» поточним символом є будь-який інший, то нічого з ним не робити (ігнорувати).

Нарешті, після того, як вхідний файл буде прочитано повністю, потрібно застосувати ще одне правило: якщо після досягнення кінця файлу поточним режимом є «примітка», то зафіксувати помилку — останню примітку не закрито.

Крім того, високоякісна програма повинна не лише повідомляти користувача про наявність помилки у вхідному файлі, але й точно вказувати номер рядка та номер символу в рядку, де помилка виникла, щоб користувачеві було легше її виправити. Тому в програмі з'являються дві нові змінні, `line` та `col` цілого типу, відповідно, номер поточного рядка та позиція в рядку. До алгоритму додаються ще два правила:

- Якщо поточний символ є символом переходу на новий рядок (незалежно від режиму), збільшити лічильник рядків `line` на одиницю, а лічильник символів у рядку `col` обнулити;
- В будь-якому іншому випадку збільшити на одиницю лічильник `col`.

Програма, як вже неодноразово зазначалося, повинна повертати операційній системі код завершення, 0 при успіху, відмінне від 0 число в разі помилки. Для цього в програмі запровадимо змінну `result`. Від початку їй присвоїмо значення 0, а у випадку помилки присвоїмо їй значення -1.

```
1  #include <stdio.h>
2  #define LEN 80
3  int main() {
4    int x, mode=1, line=0, col=0, result=0;
5    FILE *in, *out;
6    char fileName[ LEN ];
7    printf( "Ім'я␣вхідного␣файлу?␣");
8    scanf("%s", fileName);
9    in = fopen(fileName, "r");
10   if( f == NULL ) {
11     printf("Неможливо␣відкрити␣для␣читання\n");
12     return -1;
13   }
14   printf("Ім'я␣файлу␣результатів?␣");
```

```
15    scanf("%s", fileName);
16    out = fopen(fileName, "w");
17    if( out == NULL ) {
18      printf("Неможливо␣відкрити␣для␣запису\n");
19      fclose(in);
20      return -1;
21    }
22    while( !feof(in) ) {
23      x = fgetc(in);
24      if( (char) x == '\n' ) {
25        ++line;
26        col = 0;
27      }
28      else ++col;
29      if( (mode==1) && (x=='<') ) {
30        mode = 0;
31        continue;
32      }
33      if( (mode==1) && (x=='>') ) {
34        printf("%d:%d:␣>␣без␣<\n", line, col);
35        result = -1;
36        break;
37      }
38      if( mode == 1 )
39        fputc(x, out);
40      if( (mode==0) && (x=='<') ) {
41        printf("%d:%d:␣<␣у␣примітці\n", line, col);
42        result = -1;
43        break;
44      }
45      if( (mode==0) && (x=='>') ) {
46        mode = 1;
47        continue;
48      }
49    }
50    if( feof(in) && (mode == 0) ) {
51      printf("%d:%d:␣не␣закрито␣>\n", line, col);
52      result = -1;
53    }
54    fclose(in);
55    fclose(out);
56    return result;
57  }
```

Насамкінець зазначимо, що викладений в цьому розділі матеріал є першим кроком до розуміння алгоритмів роботи компіляторів, які студенти вивчатимуть в окремому курсі.

Запитання

1. Що програма, наведена в попередньому розділі, запише у файл-результат, якщо їй на вхід подати текст з незбалансованими дужками?
2. Вдосконалити розглянуту програму так, щоб вона коректно обробляла вкладені примітки, наприклад, такі:

```
ab <c<def <ghi> jkl> mno> uvw
```

10.7. Пристрої як спеціальні файли

Ключові слова: звертання до пристрою як до файлу, спеціальні потоки, стандартні потоки введення, виведення та повідомлень про помилки, `stdin`, `stdout`, `stderr`.

Для того, щоб засвоїти матеріал цього розділу, треба добре зрозуміти особливу ідеологію, що лежить в основі роботи з пристроями в операційній системі UNIX та у мові C. Основний принцип полягає в тому, що кожен приєднаний до комп'ютера пристрій (текстовий дисплей, клавіатура, принтер, модем, звукова плата) розглядається як своєрідний уявний файл. Наприклад, введення символів з клавіатури це те ж саме, що читання даних з уявного файлу, пов'язаного з клавіатурою, виведення на екран виглядає як запис у пов'язаний з екраном файл.

Ця надзвичайно вдала ідея дозволяє в однаковий спосіб обробляти введення з дискового файлу та з пристрою, такого, як клавіатура чи модем. Величезна вигода полягає в тому, що програма може «не замислюватися» над тим, звідки насправді беруться дані, з диску чи з пристрою — один і той самий текст програми підходить для обох випадків.

В заголовочному файлі `stdio.h` оголошено спеціальні змінні, в яких містяться покажчики на *стандартні потоки* введення-виведення. Найголовніші з них:

- `stdin` — стандартний потік введення, зв'язаний з клавіатурою;
- `stdout` — стандартний потік для виведення «звичайної» інформації, зв'язаний з дисплеєм;
- `stderr` — стандартний потік, також зв'язаний з дисплеєм, але призначений для виведення повідомлень про помилки.
- `stdprn` — стандартний потік виведення, пов'язаний з принтером.

Функція `printf` насправді виводить символи в стандартний потік виведення `stdout`, а функція введення `scanf` бере символи зі стандартного потоку введення `stdin`. Таким чином, наступні два оператори роблять в точності одне й те саме:

```
printf("Древляни, поляни, волиняни\n" );
fprintf(stdout, "Древляни, поляни, волиняни\n");
```

Наступні два оператори — також:

```
scanf("%d", &x);
fscanf(stdin, "%d", &x);
```

Наведемо деякі корисні поради, пов'язані з таким підходом до обробки файлів. По-перше, для друку повідомлень про помилки під час роботи програми призначено потік `stderr`. Саме ним і треба користуватися в гарно написаній програмі замість того, щоб виводити такі повідомлення функцією `printf` (в такому випадку повідомлення пішли б в інший потік `stdout`). Приклад:

```
1  int n, *p;
2  printf("Кількість елементів ");
3  scanf( "%d", &n );
4  p = (int*) malloc( n * sizeof(int) );
5  if( p == NULL ) {
6    fprintf(stderr, "Не вистачає пам'яті\n");
7    exit( -1 );
8  }
9  /* нормальна робота */
10 . . .
```

Тут повідомлення для користувача, яке супроводжує нормальну роботу програми (запит на кількість елементів масиву), виводиться функцією `printf` і потрапляє у потік `stdout`, а повідомлення про помилку (нестача пам'яті) виводиться у спеціальний потік для помилок (хоча обидва ці потоки зрештою пов'язані з одним і тим самим дисплеєм).

Ще один типовий спосіб використання файлів-пристроїв полягає в тому, що програма може, залежно від вибору користувача, виводити свої результати або на ектан, або в файл. Розглянемо текст програми:

```
1  #define LEN 80
2  int main() {
3    FILE *f;
4    char fileName[ LEN ];
5    printf("Результати␣у␣файл␣чи␣на␣екран?\n");
6    printf("ім'я␣файлу␣або␣Enter␣");
7    fgets( fileName, LEN, stdin );
8    if( strcmp(fileName, "") == 0 )
9      f = stdout;
10   else
11     f = fopen(fileName, "w");
12   if( f == NULL ) {
13     fprintf(stderr, "Неможливо␣відкрити␣файл\n");
14     exit( -1 );
15   }
16   fprintf(f, "Результати:\n");
17   /* основна частина програми */
18   . . .
19     fprintf( f, "Кінець\n" );
20   fclose( f );
21   return 0;
22 }
```

Ця програма в своїй основній частині робить деякі обчислення та, можливо, виводить великий обсяг даних (звіт про результати роботи). Програма надає користувачеві можливість обрати, як вивести звіт: надрукувати на екран чи записати у файл. Перший варіант зручний, коли користувач бажає одразу ж, у процесі роботи програми, продивитися результати, а другий — коли користувачу потрібно зберегти їх надовго.

Для того, щоб одним і тим самим оператором можна було виводити інформацію як на екран, так і в файл, використано такий прийом. В програмі запроваджено змінну `f` — потік, у який програма буде виводити свої результати. Але значенням змінної `f` може бути не лише потік, пов'язаний з дисковим файлом, але й потік `stdout`, пов'язаний з дисплеєм.

Нагадаємо, що функція `fgets` дозволяє ввести в тому числі і порожній рядок. Якщо користувач ввів ім'я файлу, робиться спроба його відкрити, відкритий потік присвоюється у змінну `f` (а якщо спроба не вдалася, програма завершується, надрукувавши повідомлення про помилку в потік помилок `stderr`). Якщо ж користувач вводить порожнє ім'я файлу, то змінній `f` присвоюється потік виведення на екран.

Всюди далі в програмі для виведення результатів використовується функція `fprintf` з потоком `f` — залежно від того, яке значення раніше було присвоєно змінній `f`, виведення піде або на екран, або у файл.

Увага! Тепер, знаючи про потік `stdin` та про функцію `fgets`, ще раз уважно перечитайте про введення текстових рядків на с. 107.

Запитання

1. Що означає вислів, що в мові C пристрої введення-виведення обробляються як файли?
2. Що означають слова `stdin`, `stdout`, `stderr`?
3. Переписати 3 раніше розроблені програми так, щоб повідомлення про помилки виводилися в спеціально призначений для цього потік.

Розділ 11

Обробка файлів, частина 2

11.1. Двійкові файли

Ключові слова: `size_t`, `fwrite`, `fread`.

Двійкові файли, на відміну від текстових, можуть містити такі коди, які не допускають відображення на екрані. Крім того, інформація у текстових файлах структурується розбиттям на рядки, пробілами між словами, табуляціями, структуру тексту видно людським оком. Двійковий файл, навпаки, це просто послідовність байтів, не призначена для зручності читання людиною, двійковий файл може обробляти лише програма, яка «знає», який сенс має той чи інший байт[1].

Для обробки двійкових файлів використовуються ті ж потоки (змінні типу покажчиків на структуру `FILE`), що і для текстових файлів (див. попередню главу). Для відкриття двійкового файлу використовується та ж функція `fopen`, а для закриття — функція `fclose`. Відрізняються лише функції введення та виведення.

Перш ніж починати описувати ці функції, треба зробити кілька вступних зауважень.

При роботі з файлами дуже часто доводиться мати справу з числами, сенсом яких є *розмір файлу чи його частини*. Звичайно ж, розмір вимірюється завжди цілим числом (причому невід'ємним), і можна було б скористатися звичайним типом **`int`**. Але створювачі мови вирішили виділити цей тип окремою назвою, щоб підкреслити його особливе призначення: `size_t` — це просто інше ім'я, запроваджене через **`typedef`**, для довгого беззнакового цілого типу.

При обробці текстових файлів функції введення спочатку читають з файлу символи, а потім розглядають їх як, наприклад, десятковий запис цілого числа і розміщують відповідне значення в пам'яті. В двійковому файлі числа зберігаються вже не в десятковому записі, а в такому ж точно вигляді, в якому вони існують в оперативній пам'яті машини. Іншими словами, записати інформацію до двійкового файлу — значить перенести на диск точну копію вмісту деякої області оперативної пам'яті, байт за байтом.

В основу обробки двійкових файлів покладено таку ідею: обмін даними між пам'яттю та файлом здійснюється блоками однакового розміру. За одну операцію виведення можна взяти з пам'яті та скопіювати на диск певну кількість блоків, за одну операцію введення можна, навпаки, кілька блоків прочитати з файлу та «покласти» в пам'ять. Важливо, що введення-виведення здійснюється лише цілими блоками (неможливо, скажімо, ввести півтора блоки).

Важливо, що блоки, які записуються у файл, повинні стояти у пам'яті підряд. Також і при читанні даних — блоки, прочитані з файлу, розташуюся у пам'яті підряд.

Для запису даних до двійкового файлу призначена функція `fwrite`, яка має чотири аргументи:

- **`void`** `*p` — покажчик на те місце в оперативній пам'яті, де починається послідовність блоків даних, яку треба записати у файл;
- `size_t b` — довжина в байтах одного блоку;
- `size_t n` — число блоків;
- `FILE *f` — покажчик на потік: до якого файлу записати дані.

[1] Поділ файлів на текстові та двійкові пов'язаний не з тим, для якої інформації призначено файл, а з тим, як цю інформацію у файлі представлено. Наприклад, файли у форматі MS Word document (`.doc`) начебто містять в собі тексти, але ці тексти закодовані таким чином, що самі файли є двійковими.

Функція бере з пам'яті, починаючи з місця, на яке вказує покажчик `p`, `n` блоків даних, кожен розміром `b` байт, та записує їх у файл, пов'язаний з потоком `f`.

Функція повертає ціле значення — число блоків, які їй вдалося записати. При успішній роботі, звичайно ж, це число повинно дорівнювати `n`, менше значення повинно свідчити про помилку у процесі запису у файл (наприклад, перевовнення диску). В підсумку, функція має прототип

```
size_t fwrite(void *p, size_t b, size_t n, FILE *f);
```

Приклади застосування цієї функції будуть наведені в наступному розділі.

Для введення даних з двійкового файлу призначена функція `fread`. Вона має ті ж аргументи, що й функція `fwrite`, але передає дані в протилежному напрямку: з потоку `f` читає `n` блоків даних, кожен розміром `b` байт, та розміщує їх в пам'яті, починаючи з місця, на яке вказує покажчик `p`.

Запитання

1. В чому відмінність двійкових файлів від текстових? Як зберігаються у двійковому та текстовому файліх, наприклад, цілі числа?
2. Що собою являє тип `size_t`?
3. Що мають на увазі, коли кажуть, що обмін даними з двійковим файлом здійснюється поблочно?
4. Призначення, аргументи функцій `fread` та `fwrite`, значення, яке вони повертають.

11.2. Прості приклади

Розглянемо приклад програми, яка записує масив дійсних чисел у файл з іменем «`data.dat`». Щоб не ускладнювати текст, пропустимо обробку помилок відкриття файлу (нагадуємо, що у «справжній» програмі такі перевірки повинні бути обов'язково).

```
1  #include <stdio.h>
2  #define N 5
3  int main() {
4    double w[ N ] =
5      { 2.0, 1.4142, 1.1892, 1.0905, 1.0443 };
6    char fileName[] = "data.dat";
7    FILE *out;
8    int k;
9    out = fopen( fileName, "w" );
10   k = fwrite( w, sizeof(double), N, out );
11   printf( "У␣файл␣записано␣%d␣чисел", k );
12   fclose( out );
13   return 0;
14 }
```

Тут кожен блок даних, що записуються до файлу, є числом типу **`double`** — елементом масиву `w`. Тому при виклику функції `fwrite` їй передаються такі значення аргументів:

1. Покажчик на початок області даних — це адреса масиву `w`;
2. Розмір одного блоку даних — це розмір одного елементу масиву, тобто розмір одного числа типу **`double`**;
3. Кількість блоків даних — це кількість елементів масиву `w`, тобто константа `N`.

Іншими словами, виклик функції `fwrite` означає буквально таке: з того місця в оперативній пам'яті, де починається масив `w`, взяти `N` блоків даних (всі блоки однакового розміру, такого, як одне число дійсного типу), та записати їх до файлу, зв'язаного з потоком `out`.

Число успішно записаних блоків (елементів масиву) присвоюється у змінну `k`. При нормальній роботі програма має повідомити, що у файл записано 5 чисел.

Розглянемо інший приклад: нехай програма також записує до файлу числа, але не весь масив за одну операцію, а по одному:

```
1  #include <stdio.h>
2  #define N 5
3  int main() {
4    double w[ N ] =
5      { 2.0, 1.4142, 1.1892, 1.0905, 1.0443 };
6    char fileName[] = "data.dat";
7    FILE *out;
8    int i;
9    out = fopen( fileName, "w" );
10   for( i = 0; i < N; ++i )
11     fwrite( &(w[i]), sizeof(double), 1, out );
12   fclose( out );
13   return 0;
14 }
```

Тут функція запису викликається в циклі 5 разів, при кожному виклику вона записує у файл вміст чергового елементу масиву `w`: аргументи означають, що записувати треба дані, починаючи з тієї адреси, де лежить ця змінна, записати треба один блок такого розміру, який займає в пам'яті дійсне число.

Тепер розглянемо програму, яка читає та роздруковує масив дійсних чисел з файлу «`data.dat`», створеного попередньою програмою.

```
1  #include <stdio.h>
2  #define N 5
3  int main() {
4    double w[ N ];
5    char fileName[] = "data.dat";
6    FILE *in;
7    int k, i;
8    in = fopen( fileName, "w" );
9    k = fread( w, sizeof(double), N, in );
10   printf( "З␣файлу␣прочитано␣%d␣чисел", k );
11   for( i = 0; i < k; ++i )
12     printf( "%lf\n", w[i] );
13   fclose( in );
14   return 0;
15 }
```

«Серцем» програми є виклик функції `fread`, якій наказано (див. значення аргументів) з потоку `in` прочитати `N` (тобто 5) блоків даних, кожен такого ж розміру, як дійсне число, а прочитані дані розмістити в пам'яті там, де починається масив `w`. Отже, програма читає з файлу 5 чисел та присвоює їх елементам масиву. Значенням змінної `k` при цьому стає кількість успішно прочитаних блоків даних (тобто чисел).

Розглянемо також програму, яка читає та роздруковує числа з файлу один за одним:

```
1  #include <stdio.h>
2  int main() {
3    double x;
4    char fileName[] = "data.dat";
5    FILE *in;
6    int k=0;
7    in = fopen( fileName, "r" );
8    while( !feof(in) ) {
9      fread( &x, sizeof(double), 1, in );
10     printf("%lf\n", x);
```

```
11      ++k;
12    }
13    printf("всього␣%d␣чисел\n", k);
14    fclose( in );
15    return 0;
16  }
```

Ця програма читає числа по одному, одне число за одну операцію. Аргументи функції `fread` означають: взяти з файлу (потоку) `in` один блок даних такого ж розміру, як дійсне число, та покласти в оперативну пам'ять, в ту комірку, де розташована змінна `x`. Цикл означає, що програма буде повторювати вказану операцію доти, поки не закінчиться файл, а кожне значення, прочитане з файлу та занесене до змінної `x`, буде друкуватися на екрані.

Запитання

1. Написати програму, яка запитує у користувача ціле число n, динамічно виділяє пам'ять під масив з n дійсних чисел, вводить його елементи і потім записує весь масив у файл.
2. Змінити попередню програму так, щоб вона сортувала масив перед записом у файл.
3. Написати програму, яка запитує у користувача ціле число n та записує до файлу n перших чисел Фібоначчі.
4. Написати програму, яка читає числа Фібоначчі з файлу, створеного попередньою програмою, та записує в інший файл частки від ділення двох сусідніх чисел, тобто величини $\frac{F_i}{F_{i+1}}$.

11.3. Файли структур

В розглянутих вище прикладах кожен блок даних, якими програма обмінювалася з потоком, був числом типу **double**. Це один з найпростіших способів роботи з двійковими файлами — за кожну операцію введення чи виведення передавати дані певного простого типу. Разом з тим, величезну роль в практичних застосуваннях має випадок, коли в ролі блоків виступають дані структурних типів.

В розділі 9.3 було сказано, що масиви структур є програмними моделями таблиць. Файли, що містять в собі певну кількість записаних підряд структур, є, фактично, також таблицями, тільки призначеними для довгого зберігання. Файли, що складаються з записів однакової структури, є найпростішим різновидом баз даних.

Наведемо приклад програми, що записує у файл базу даних — список студентів з оцінками по програмуванню. Програма працює в циклі: вона щоразу запитує у користувача прізвище та оцінки студента з теорії та за лабораторні роботи, записує сформований про цього студента запис до файлу та запитує у користувача, продовжувати чи завершувати роботу. У першому випадку процес повторюється для наступного запису.

Для економії місця та щоб не відволікати увагу читача надмірними подробицями, знов пропускаємо обробку можливих помилок введення-виведення.

```
1  #include <stdio.h>
2
3  #define FNAME_LEN 80
4  #define SURNAME_LEN 20
5
6  typedef struct tag_student {
7    char surname[SURNAME_LEN];
8    int prog_theor;
9    int prog_lab;
10 } tstudent;
11
12 int main() {
```

```
13    char fileName[FNAME_LEN];
14    FILE *out;
15    tstudent s;
16    int ans, n=0;
17    printf("Ім’я␣файлу?␣");
18    fgets(fileName, FNAME_LEN, stdin);
19    out = fopen( fileName, "w" );
20    do {
21      printf("Ім’я?␣");
22      printf("Прізвище?␣");
23      fscanf(s.surname, SURNAME_LEN, stdin);
24      printf("Оцінки␣з␣теорії␣та␣лаб.робіт?␣");
25      scanf("%d%d", &(s.prog_theor), &(s.prog_lab));
26      fwrite( &s, sizeof(tstudent), 1, out);
27      ++n;
28      printf("Продовжити␣(1␣так,␣0␣ні)?␣");
29      scanf("%d", &ans);
30    } while( ans );
31    printf("У␣файлі␣%d␣записів\n", n);
32    fclose( out );
33    return 0;
34  }
```

Як видно, блоком даних є структура типу `tstudent`, за кожну операцію до файлу виводиться рівно один блок — вміст змінної `s`. Після завершення програми дані у файлі будуть сформовані так:

Щоб проілюструвати читання структур з файлу, наведемо програму, яка читає базу даних по студентах, сформовану попередньою програмою, та роздруковує її вміст на екрані у вигляді таблиці з трьох колонок.

```
1  #include <stdio.h>
2
3  #define FNAME_LEN 80
4  #define SURNAME_LEN 20
5
6  typedef struct tag_student {
7    char surname[SURNAME_LEN];
8    int prog_theor;
9    int prog_lab;
10 } tstudent;
11
12 int main() {
13   char fileName[FNAME_LEN];
14   FILE *in;
15   tstudent s;
16   int n=1;
17   printf("Ім’я␣файлу?␣");
18   fgets(fileName, FNAME_LEN, stdin);
19   in = fopen( fileName, "r" );
20   while( !feof(in) ) {
21     fread( &s, sizeof(tstudent), 1 );
22     printf("%3d%30s%10d%10d\n",
23         n, s.surname, s.prog_theor, s.prog_lab);
24     ++n;
25   }
26   fclose( in );
27   return 0;
28 }
```

Запитання

1. Вдосконалити дві наведені в цьому розділі програми так, щоб вони коректно обробляли можливі помилки.
2. Написати програму, яка записує у файл послідовність структур типу `tPoint` (с. 121) — таблицю координат точок одиничного кола.
3. Написати програму, яка читає з файлу послідовність координат точок та знаходить з них ту, що лежить найближче до точки, координати якої користувач вводить з клавіатури.
4. Написати програму, яка читає з файлу послідовність координат точок та серед тих із них, у яких $-0.3 < x < 0.3$, знаходить таку, що лежить найближче до точки, координати якої вводить з клавіатури користувач.

11.4. Переміщення по файлу

Ключові слова: `fseek`, `ftell`.

Обробка текстових файлів частіше за все полягає у послідовному, літера за літерою, читанні одного файлу та такому ж послідовному записі результатів в інший файл. При обробці двійкових файлів у багатьох практичних задачах виникає потреба читати чи записувати файл не підряд, а в довільному порядку, час від часу переміщуючись по ньому вперед і назад.

Уявімо, що файл — це довга стрічка, поділена на комірки, в кожній з яких може зберігатися один байт. По стрічці переміщується голівка читання-запису. При звичайних (послідовних) операціях читання та запису голівка, прочитавши чи записавши комірку навпроти якої стоїть, переміщується на одну комірку вперед. Разом з тим, в мові C існують функції, які дозволяють рухати голівку на довільну відстань вперед та назад по файлу. Якщо встановити голівку на певну комірку, наступна операція читання чи запису буде відноситися саме до цієї комірки.

Функція `fseek` доволі універсальна: вона дозволяє встановити голівку на n-й байт, рахуючи від початку файла, на n-й байт від кінця файлу, або перемістити голівку на n байтів (вперед чи назад) відносно її поточного місця. Функція має три аргументи:

- покажчик на потік — в якому файлі встановлювати голівку;
- ціле число — на яку відстань переміщувати голівку;
- ціле число, що означає режим переміщення голівки (рахувати відстань від початку, від кінця файлу, або від поточної позиції голівки).

Прототип цієї функції:

```
int fseek( FILE *f, long offset, int whence );
```

Нагадаємо, що тип **`long`** — це довге ціле число (зі збільшеною розрядною сіткою), хоча на багатьох сучасних платформах має таку ж розрядність, як і тип **`int`**. Позначаючи константи довгого типу, потрібно наприкінці приписати літеру `L`, наприклад константа `0` — це число 0 в типі **`int`**, а константа `0L` — число 0 в типі **`long`**. Для введення-виведення значень даного типу слугує специфікатор `%ld`.

Для режимів переміщення (третій аргумент) є константи з символічними іменами:

- `SEEK_SET` — від початку файлу;
- `SEEK_CUR` — від поточного розташування голівки;
- `SEEK_END` — від кінця файлу.

Наприклад, перший з наведених нижче викликів функції переставляє голівку на початок файлу, другий — на кінець файлу, третій рухає голівку на п'ять байтів вперед, четвертий рядок рухає її на сім байтів назад.

```
FILE *f;
fseek( f, 0L,  SEEK_SET );
fseek( f, 0L,  SEEK_END );
fseek( f, 5L,  SEEK_CUR );
fseek( f, -7L, SEEK_CUR );
```

При успішному виконанні функція `fseek` повертає значення 0, в іншому випадку значення −1. Неуспіх може відбутися, наприклад, при спробі поставити голівку перед початком файлу.

Одне з типових застосувань функції `fseek` — читання k-го від початку файлу запису. Повернемося до розгляду нашого прикладу з базою даних про оцінки студентів. Наступна програма дозволяє *відредагувати* k-й за порядком запис в базі — ввести інші оцінки. Щоб не займати місця, не наводимо оголошення структурного типу `student` та макросів — вони залишаються такими ж, як і в попередніх прикладах.

```
1  . . .
2  int main() {
3    char fileName[ FNAME_LEN ];
4    FILE *f;
5    tstudent s;
6    int k, r;
7    printf("Файл␣бази␣даних?␣");
8    fgets( fileName, FNAME_LEN, stdin );
9    f = fopen( fileName, "w+" );
10   while( 1 ) {
11     printf("Номер␣запису␣(-1␣вихід)␣");
12     scanf("%d", &k);
13     if( k == -1 ) break;
14     r = fseek(f, (long) k * sizeof(tstudent),
15         SEEK_SET);
16     fread( &s, sizeof(tstudent), 1 );
17     printf("Старий␣запис␣в␣базі\n");
18     printf("%30s%10d%10d",
19         s.surname, s.prog_theor, s.prog_lab);
20     printf("Введіть␣виправлені␣оцінки:␣");
21     scanf("%d%d", &(s.prog_theor), &(s.prog_lab));
22     fseek(f, (long) k * sizeof(tstudent),
23         SEEK_SET);
24     fwrite( &s, sizeof(tstudent), 1 );
25   }
26   fclose( f );
27   return 0;
28 }
```

Програма працює у циклі. Щоразу програма запитує у користувача, номер запису, який він бажає відредагувати. Якщо користувач вводить значення −1, то цикл розривається оператором **break** та програма завершується.

Якщо ж введене користувачем число `k` відрізняється від −1, програма робить спробу поставити голівку читання-запису на те місце у файлі, де починається запис під номером `k`. Очевидно, `k`-й запис починається на байті, номер якого дорівнює добутку `k` на довжину одного запису (див. другий аргумент функції `fseek`).

Один запис читається в буферну змінну `s`. Вміст її полів друкується на екран, після чого користувач вводить нові значення двох оцінок, ці значення заносяться в поля тієї ж структурної змінної `s`. Оскільки попередня операція читання запису пересунула голівку читання-запису, її знов треба переставити на початок `k`-го запису, для цього ще раз викликається функція `fseek` з тими ж значеннями аргументів. Нарешті, викликається функція `fwrite`, яка у файл на місце `k`-го запису записує вміст змінної `s`.

В мові C є також функція `ftell`, що дозволяє дізнатися, на якій позиції в даний момент

стоїть голівка. Функція має один аргумент типу покажчика на потік, повертає довге ціле число — номер байту від початку файлу, на якому зараз стоїть голівка.

За допомогою функцій `fseek` та `ftell` легко дізнатися розмір файлу. Для цього достатньо спочатку поставити голівку на кінець файлу, а потім дізнатися її поточну позицію:

```
FILE *f;
long l;
char fileName = "data.dat";
f = fopen( fileName, "r" );
fseek( f, 0L, SEEK_END );
l = ftell( f );
printf( "Довжина␣файлу␣%ld␣байтів\n", l );
```

Запитання

1. Як переміститися на початок файлу? На 10-й від початку байт? На 10-й від кінця? На 10 байтів вперед та назад відносно поточної позиції?
2. Як з програми дізнатися, на якому місці файлу відбувається читання-запис?
3. Вдосконалити наведену в даному розділі програму обробки списку студентів так, щоб вона коректно обробляла можливі помилки.
4. Вдосконалити програму так, щоб вона дозволяла додавати до списку нових студентів.

11.5. Способи організації файлів

Ключові слова: заголовок файлу, формат файлу.

Хоча в мові C є ще доволі багато функцій для роботи з файлами, викладених вище засобів достатньо, щоб запрограмувати, в принципі, будь-яку задачу обробки файлів. Тепер потрібно зосередити увагу не на вивченні нових функцій (вони нічого суттєвого не додають до вже вивчених), а на способах роботи з файлами. В практиці програмування накопичено чимало підходів, що за кілька десятиліть стали фактичним стандартом та які неодмінно повинен знати кожен професіонал. Їх розгляд будемо проводити на прикладах.

Розглянемо програму, яка запитує у користувача рядки тексту та записує їх у двійковий файл. Програма не знає заздалегідь, скільки рядків введе користувач — вона завершує свою роботу, коли замість чергового рядка користувач введе крапку. Нехай програма запише до файлу кількість всіх введених рядків.

Зрозуміло, що програма може не запам'ятовувати всі раніше введені рядки — кожен раз вона має пам'ятати лише один, щойно введений, рядок. Тому в програмі оголосимо лише один масив символів — назвемо його буфером. Під буфер потрібно зарезервувати достатньо багато місця, щоб введений користувачем текст завідомо вмістився. Нехай масив матиме довжину 256 байт (255 літер та один байт для нуль-символа). Роблячи буфер великим, ми розраховуємо на те, що більшість введених користувачем рядків матимуть значно меншу довжину (наприклад, довжина рядка у книзі зазвичай становить біля 60 символів).

Виникає питання, як розміщувати текстові рядки у файлі. Можна було б зберігати у файлі блоки розміром по 256 байт, в кожному з яких поміщався б цілий буфер. Але це призвело б до дуже неекономного використання дискового простору — в кожному з таких блоків корисною інформацією було б зайнято лише перші кілька десятків байт.

Для подібних випадків добре підходить такий прийом. Будемо записувати у файл кожен рядок разом з його довжиною. Для кожного введеного користувачем рядка спочатку запишемо у файл його довжину (число цілого типу, що займає 2 байти), а потім власне текст (разом з завершальним нуль-символом). Про такий спосіб зберігання даних кажуть, що блоки даних у файлі мають змінну довжину та супроводжуються префіксами.

Крім того, потрібно визначити, як записати до файлу кількість всіх рядків. Зробимо так: створюючи файл даних, на самому початку запишемо туди ціле число — будь-яке. Таким

чином ми зарезервуємо на початку файлу два вільні байти. Після того, як користувач введе останній рядок, програма поверне голівку читання-запису на початок файлу та у зарезервовані перші два байти запише кількість рядків. Це приклад ще одного дуже важливого стандартного підходу до обробки файлів — заголовку файлу.

Отже, файл даних буде мати таку будову:

- Заголовок файлу — ціле число n;
- n записів змінної довжини, кожна з яких має таку будову:
  - префікс елементу — ціле число m;
  - m байтів, що складають рядок (разом з нуль-символом).

Зазначимо, що хоча у файлі зберігаєтья начебто текстова інформація, але спосіб роботи з нею та спосіб організації даних у файлі означають, що це не текстовий, а двійковий файл (зокрема, у файлі зберігаються цілі числа у двійковому зображенні).

Текст програми (без обробки помилок) наведено нижче:

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

#define FNAME_LEN 80
#define BUFFER_LEN 256

int main() {
  char fileName[FNAME_BUF];
  char buf[BUFFER_LEN];
  int n=0, m;
  FILE *f;
  printf("Ім'я файлу? ");
  fgets( fileName, FNAME_LEN, stdin );
  f = fopen( fileName, "w" );
  printf("Вводіть рядки, після кожного Enter\n");
  printf("Для виходу - рядок з однієї крапки\n");
  while( 1 ) {
    fgets( buf, BUFFER_LEN, stdin );
    if( strcmp(buf, ".") == 0 ) break;
    m = strlen( buf ) + 1;
    fwrite( &m, sizeof(int), 1, f );
    fwrite( buf, m, 1, f );
    ++n;
  }
  fseek( f, 0L, SEEK_SET );
  fwrite( &n, sizeof(int), 1, f );
  fclose( f );
  return 0;
}
```

Цікаво розглянути також програму, яка читає файл описаної вище будови та копіює всі рядки з нього в оперативну пам'ять. Програма заздалегідь (до того, як почне читати файл) не може знати, ні скільки в ньому буде рядків, ні довжину кожного з них. Тому пам'ять для рядків потрібно виділяти динамічно. Конкретніше, потрібен динамічний масив покажчиків на динамічні масиви символів — цю роль в програмі виконує змінна `textLines`.

Відкривши файл для читання, програма в першу чергу читає його заголовок — ціле число n, кількість рядків тексту, що записані у файлі далі. Знаючи це число, програма виділяє пам'ять під масив покажчиків на рядки (n елементів). Підкреслимо, що на цьому етапі створюються лише покажчики-елементи масиву, але вони поки що нікуди не показують — пам'ять під рядки, на які вони мають вказувати, ще не виділялася.

Далі n разів повторюється така дія. Програма читає з файлу ціле число m — довжину наступного рядка, виділяє пам'ять під масив символів довжиною m, присвоює i-му елементу таблиці покажчиків `textLines` адресу цього масиву, зчитує з файлу m байтів та заносить їх у масив.

Таким чином, по завершенні описаного процесу весь розбитий на рядки текст, що міститься у файлі, переноситься у пам'ять. Кожному рядку відповідає свій елемент таблиці покажчиків `textLines`. Після якоїсь обробки цих рядків (в тексті програми показано коментарем) потрібно звільнити динамічну пам'ять та закрити файл.

```
1  #include <stdio.h>
2
3  #define FNAME_LEN 80
4  #define BUFFER_LEN 256
5
6  int main() {
7    char fileName[FNAME_BUF];
8    char **textLines;
9    int n, m, i;
10   FILE *f;
11   printf("Ім'я␣файлу?␣");
12   fgets( fileName, FNAME_LEN, stdin );
13   f = fopen( fileName, "r" );
14   fread( &n, sizeof(int), 1, f );
15   textLines = (char**) malloc(n * sizeof(char*));
16   for( i=0; i<n; ++i ) {
17     fread( &m, sizeof(int), 1, f );
18     textLines[i] = (char*)malloc(m*sizeof(char));
19     fread( textLines[i], m, 1, f );
20   }
21   fclose( f );
22   /* обробка текстових рядків в пам'яті */
23   for( i = 0; i < n; ++i )
24     free( textLines[i] );
25   free( textLines );
26   return 0;
27 }
```

Запитання

1. Вдосконалити наведені в цьому розділі програми так, щоб вони обробляли можливі помилки.
2. Написати програму, яка обробляє файл текстових рядків, дописуючи до нього нові рядки (увага: потрібно не лише дописати рядки в кінець файлу, але й змінити значення лічильника в його заголовку).
3. Написати програму, яка читає файл текстових рядків та в такому ж форматі переписує в інший файл лише ті з них, які мають парну кількість символів.
4. Написати програму, яка друкує рядок під заданим номером (номер вводиться користувачем) з файлу текстових рядків.
5. Написати програму, яка зберігає у двійковому файлі прямокутну таблицю дійсних чисел: нехай в заголовку файлу містяться кількість рядків та стовпчиків таблиці.
6. Вдосконалити програму обробки списку студентів так, щоб на кожного студента крім прізвища та оцінок зберігалася також характеристика — текст заздалегідь невідомої довжини.

11.6. Підсумковий огляд

Файл — це набір даних на зовнішньому пристрої (як правило, диску). Файл має ім'я, за яким його можна на диску знайти.

Файли дозволяють зберігати великі обсяги даних (об'єм диску набагато перевищує розмір оперативної пам'яті). Крім того, якщо змінні, які обробляє програма, знищуються в оперативній пам'яті одразу ж при завершенні програми, то файл дозволяє даним зберігатися в проміжках між завершенням програми, що їх обробляє, та наступним її запуском.

Більш того, файли дозволяють один і той самий набір даних обробляти кількома різними програмами: одна програма може записати результати своєї роботи у файл, інша програма може прочитати дані з цього файлу, зробити над ними деякі дії, та знов записати у файл, і так далі.

На відміну від даних в оперативній пам'яті, файли легко змінюють розмір — до кінця файлу можливо дописувати нові дані, можна робити і протилежне — відсікати від файлу кінець.

З іншого боку, якщо дані в оперативній пам'яті мають абсолютно довільний порідок доступу (можна звертатися в будь-якому порядку до будь-яких змінних чи елементів масивів), то файлам притаманний послідовний доступ. Всі операції з даними у файлі здійснюються лише у поточній позиції голівки читання та запису, в кожен момент часу голівка оглядає один певний байт файлу. Щоб обробляти дані з різних місць файлу, потрібно застосовувати функцію переміщення голівки.

На диску файл вирізняється з-поміж інших файлів за іменем, але програма звертається до даних файлу не по імені, а через спеціальну допоміжну змінну — потік. Потік відіграє роль посередника між програмою та фізичним пристроєм (диском). Єдиний момент, коли програма згадує ім'я файлу, під яким він записаний на диску, це момент відкриття — коли створюється потік, пов'язаний з дисковим файлом. Всі подальші операції з файлом здійснюються лише через потік.

Файл можна відкривати в різних режимах: лише для читання, лише для запису (в тому числі для дописування в кінець чи для перезапису з початку), для читання та запису.

Розрізняються способи роботи з двійковими та з текстовими файлами. Треба розуміти, що це в першу чергу не два різні типи файлів, а два способи їх обробки. При обробці текстових файлів користуються аналогами функцій, що друкують текст на екран та читають текст з клавіатури. Тим більше, що екран, клавіатура та взагалі будь-які пристрої, згідно ідеології мови C, також розглядаються як особливі пристрої. Між іншим, професійно написана програма має розрізняти потік для виведення звичайних текстових повідомлень та потік для виведення повідомлень про помилки (хоча зазвичай обидва пов'язані з одним і тим самим дисплеєм).

Робота з двійковими файлами полягає в тому, щоб записувати чи читати певну кількість блоків даних однакового розміру. Програміст, який складає програму для обробки файлу, сам визначає, яку будову матиме файл — як розташовуються в ньому блоки даних.

На практиці часто застосовують два зручних прийоми: заголовок файлу (деякі дані, що записуються на початку файлу та містять деяку інформацію, важливу для подальшої обробки всього файлу, особливо — кількість записів у файлі) та префікс елементу (інформація, записана перед блоком даних, важлива для подальшої обробки саме цього блока, наприклад, довжина текстового рядка).

Розділ 12

Модульність

Досі в цьому посібнику розглядалися лише прості та невеликі за обсягом програми, які містили всього по декілька функцій. Текст такої програми розміщувався в одному файлі. Програми, що вирішують реальні практичні задачі, мають величезний обсяг та складаються з тисяч функцій. Досвід показує, що записувати всі функції програми в один файл незручно, оскільки зі збільшенням програми в такому файлі стає важко орієнтуватися, а також через те, що при цьому унеможливлюється спільна робота кількох програмістів. Тому грамотна технологія програмування полягає в тому, щоб розбивати велику програму на частини — так звані модулі, кожен з яких складається з порівняно невеликої кількості функцій. В першому наближенні модуль являє собою окремий файл, в якому містяться тіла декількох функцій.

Переваги модульності, однак, зовсім не вичерпуються згаданою зручністю від розбиття великої кількості функцій на невеликі групи. Програміст може написати такий набір функцій, який може стати корисним для інших програмістів — коли ці функції зроблено достатньо гнучкими та універсальними, щоб їх можна було вставляти в чужі програми. Як наслідок, модулі можуть бути самостійним різновидом товару на ринку програмного забезпечення. Іншими словами, робота програміста може полягати не лише в тому, щоб виготовляти програми для кінцевого споживача, але й в тому, щоб писати модулі для інших програмістів, які будуть вставляти їх у свої програми.

Крім того, до складу одного програмного продукту цілком можуть входити модулі, написані різними мовами програмування. На початку цього посібника ми вже згадували про те, що мови програмування бувають спеціалізованими: кожною мовою найзручніше вирішуються задачі своєї певної області. Якщо ж треба вирішити комплексну задачу, яка розпадається на частини, що належать до різних областей, то найкраще кожну підзадачу реалізувати окремим модулем, написаним найзручнішою для цієї підзадачі мовою.

Отже, модульність стала одним з найвизначніших винаходів в історії програмування і досі залишається наріжним каменем сучасних технологій програмування. Модульність — це не лише суто технічний прийом, а ще й ключовий елемент *культури програмування*, вона займає центральне місце в парадигмі і, без перебільшення, має світоглядне значення.

12.1. Основні ідеї

Щоб добре зрозуміти принцип побудови багатомодульної програми, варто спочатку уявити деяку достатньо велику програму, яка вся міститься в одному файлі. Згідно попереднього матеріалу посібника, вона містить декілька функцій, кожна з яких може викликати будь-яку іншу функцію. Оскільки будь-яка функція повинна бути оголошена перед першим викликом, а кожна функція програми може викликати будь-яку іншу функцію, на самому початку програми повинні стояти оголошення усіх функцій (крім `main`).

Тепер уявімо, що цю ж програму розбито на кілька модулів, тобто тіла декількох функцій перенесено в один файл, тіла кількох інших функцій — у другий, і т.д. Функція, розміщена в одному модулі, може викликати функцію, що міститься у іншому модулі. А це означає, що на початку кожного модуля повинні міститися оголошення (прототипи) всіх *зовнішніх функцій* — функцій з інших модулів, які викликаються з цього модуля.

Прототипи зовнішніх функцій, звичайно ж, можна було б вписати в текст модуля вручну, але при великій кількості функцій це було б надто незручно. Набагато зручніше виносити прототипи функцій в окремий *заголовочний* файл (файл з розширенням `.h`) та *підключати* цей файл (за допомогою директиви **`#include`**) до кожного модуля, який має намір використовувати ці функції.

Отже, створюючи кожен модуль, програміст в один файл записує тіла кількох функцій (такий файл має розширення `.c`), а в інший, заголовочний, з розширенням `.h`, — лише прототипи цих функцій. Про заголовочний файл часто кажуть, що він містить *інтерфейс* модуля, а про `.c`-файл — що він є *реалізацією* модуля.

Трансляція багатомодульної програми має важливу особливість. Компілятор обробляє не всю багатомодульну програму як одне ціле, а окремо модуль за модулем. Модулі компілюються незалежно один від одного, іншими словами, компілятор, поки компілює один модуль, «не знає» про існування інших модулів. Наприклад, деяка функція модуля `aaa.c` викликає функцію **`int`** `f(`**`int`**`)` з модуля `bbb.c`. Тоді компілятор, поки обробляє модуль `aaa.c` знає лише ім'я функції, що викликається (воно відомо з прототипу), але не «підглядає» у файл `bbb.c`, щоб дізнатися тіло цієї функції.

В результаті такої роздільної компіляції кожного `.c`-файлу виходить так званий *об'єктний* файл (зазвичай він має таке ж ім'я, що й відповідний `.c`-файл, але з розширенням `.obj`). В об'єктному файлі операції над даними та оператори управління вже перекладено у машинні коди, але виклики функцій поки що залишено, як прийнято говорити, *нерозв'язаними* — наприклад, об'єктний файл `aaa.obj` містить виклик функції **`int`** `f(`**`int`**`)`, який поки що неможливо перетворити на машинну команду переходу до тіла функції, бо модуль `aaa.obj` нічого не знає про об'єктний файл `bbb.obj`, в якому міститься це тіло.

Нарешті, після того, як модулі відкомпільовано, спрацьовує ще одна спеціальна програма — *зв'язувач*, або *редактор зв'язків*. Спрощено кажучи, він бере всі об'єктні файли, що входять до складу програми, та об'єднує їх в один *виконуваний файл* (тобто файл, готовий для виконання на машині), розв'язуючи при цьому посилання на імена функцій — замінюючи виклик функції за іменем, що міститься в одному модулі, на конкретну адресу її тіла, взятого з іншого модуля.

Увага! Мова C сама по собі призначена для того, щоб писати нею модулі, і тільки для цього. Опис того, з яких модулів складається багатомодульна програма, здійснюється не за допомогою мови програмування, а спеціальними засобами, розгляд яких виходить за межі даної книги. Частина з них розглядається на лабораторних роботах.

12.2. Включення файлів

Вже з перших кроків у вивченні мови C в кожній своїй програмі програміст використовує директиву **`#include`**. В главі про основи мови C було сказано коротко, що ця директива дозволяє *підключити* заголовочний файл до своєї програми. В цьому розділі сенс цієї директиви та способи її використання будуть викладені детально.

Синтаксис директиви такий: після слова **`#include`** йде ім'я файлу (з розширенням), взяте або в подвійні лапки, або в в кутові дужки. Іншими словами, директива має одну з двох форм:

```
#include <filename.h>
#include "filename.h"
```

Перша форма каже транслятору, що файл, ім'я якого вказано в кутових дужках, треба шукати в спеціальній директорії, призначеній для стандартних заголовочних файлів. Саме тому в кутових дужках вказують імена таких заголовочних файлів, як `stdio.h`, `math.h` та інших, які поставляються разом з транслятором. Друга форма директиви означає, що транслятор повинен шукати файл в директорії для користувацьких заголовочних файлів (можливо, зокрема, у тій самій директорії, що й текст модуля).

Сенс директиви пояснимо спочатку спрощено. Зустрівши будь-де в тексті директиву `include`, транслятор має знайти файл з відповідним іменем, та підставити весь вміст цього файлу замість директиви.

Покажемо це на прикладі. Нехай є файл `mydecl.h` з таким змістом:

```
typedef unsigned long int ulint;
#define MAX_LEN 32
```

```
ulint f( int* k, ulint x );
```

та ще один файл з іменем `mycode.c`:

```
void g( int );
#include "mydecl.h" /* !!! */
int main() {
  int m[ MAX_LEN ];
  ulint s, r=0;
  s = ulint f( m, r );
  . . . /* ще якісь оператори */
  return 0;
}
```

Коли компілятор буде обробляти файл `mycode.c`, то, прочитавши другий рядок, він знайде файл `mydecl.h` та вставить його на місце директиви. Отже, після такої підстановки компілятор «побачить», ніби файл `mycode.c` виглядає так:

```
void g( int );
typedef unsigned long int ulint;
#define MAX_LEN 32
ulint f( int* k, ulint x );
int main() {
  int m[ MAX_LEN ];
  ulint s, r=0;
  s = ulint f( m, r );
  . . . /* ще якісь оператори */
  return 0;
}
```

Наведене вище пояснення роботи директиви насправді трохи спрощене. Більш точно процес виглядає так: транслятор читає файл рядок за рядком; коли транслятор бачить в тексті директиву **#include**, він тимчасово припиняє обробляти поточний файл, шукає той файл, ім'я якого вказано в лапках після слова`include`, та починає так само, рядок за рядком, опрацьовувати його; дійшовши до кінця заголовочного файлу, транслятор знов повертається до недочитаного файлу та продовжує обробляти його з того самого місця, де припинив.

Таким чином, ми розібрали синтаксис та семантику директиви включення. Перейдемо до найважливіших та водночас тонких питань прагматики — способів красивого застосування даного засобу. Одразу слід зазначити: гнучкість та універсальність мови C настільки великі, що мова однаково добре дозволяє як красиві, так і дуже некрасиві конструкції та програмістські рішення на основі директиви **#include**. Як було вже неодноразово зазначено раніше з приводу інших елементів мови C, транслятор не бере на себе турботу вберігати від поганого стилю та некрасивих рішень — за стрункість і логічність програмного тексту відповідає програміст.

Стандарт та означення мови C не містять правил щодо імен та розширень заголовочних файлів — в принципі в директиві підключення можна вказати ім'я файлу з будь-яким розширенням (головне, щоб цей файл містив текст, зрозумілий для транслятора, тобто текст мовою C). Але за усталеною традицією файлам, спеціально призначеним для включення в інші файли, надають розширення `.h` — від англ. «header» (заголовок).

Як вже неодноразово зазначалося вище, в заголовочних файлах прийнято розміщувати *оголошення*, потрібні в основному тексті програми. Але важливо розуміти, що використання `.h`-файлів саме для оголошень — не синтаксичне правило, а традиція. Транслятор не перевіряє, знаходяться у підключеному файлі оголошення чи якісь інші конструкції мови C, — єдине, що транслятор вимагає, це щоб у підключеному файлі був деякий текст, правильний з точки зору граматики мови C.

Тому, в принципі, правила мови дозволяють і такий трюк, який, на жаль, часто приваблює початківців: розбити довгий програмний текст на кілька частин, кожну оформити в окремому

файлі, а в ще одному файлі зібрати ці частини до купи за допомогою директиви `include`. Розглянемо приклад:

Файл `part1.h`:

```
int main() {
  int n, i, j;
  printf( "Введіть␣розмір␣трикутника␣" );
  scanf( "%d", &n );
  for( i = 1; i < n; ++i ) {
    for( j = 1; j <= i-1; ++j )
```

Файл `part2.h`:

```
      printf( "␣" );
    for( j = 1; j <= n-i+1; ++j )
      printf( "*" );
    printf( "\bck{}n" );
  }
  return 0;
}
```

Файл `prog.c`

```
#include <stdio.h>
#include "part1.h"
#include "part2.h"
```

Транслятор, обробивши включення файлів `part1.h` та `part2.h`, отримає синтаксично правильний текст програми з прикладу на с. 56. Але, незважаючи на те, що створені в такий спосіб програми нормально компілюються і працюють, **в жодному разі не можна застосовувати цю «технологію»**, оскільки вона грубо суперечить нормам красивого стилю програмування.

Дуже типова помилка початківців полягає в тому, що модульність підміняють таким включенням шматків програмного коду з різних файлів. Насправді це не має нічого спільного з модульністю. Модулі, як вже було сказано вище, повинні компілюватися незалежно один від одного, щоб програма як ціле могла збиратися з цих частин. А в описаному тут помилковому підході всі частини програмного коду компілюються разом — завдяки директивам включення компілятор «побачить» один суцільний програмний текст.

Увага! Не все те, що дозволено граматикою мови та нормально пропускається транслятором, є припустимим з точки зору культури програмування.

Увага! В заголовочному файлі можна розміщувати: прототипи функцій, оголошення типів (особливо структур), означення макросів — всі такі оголошення, які використовуються на етапі компіляції. В жодному разі не слід розміщувати в заголовочному файлі програмний код, що стосується етапу виконання програми — тобто тіла функцій.

12.3. Проблема подвійного включення

Використовуючи викладені вище засоби, можна зіткнутися з серйозними незручностями. Спочатку опишемо та проілюструємо сутність проблеми, а потім запровадимо зручний засіб для їх усунення.

На практиці можливі ситуації, коли один заголовочний файл прямо чи непрямо підключається до якогось модуля два або більше разів. Це означає, що кожне оголошення з цього заголовочного файлу потрапить в текст програми кілька разів, а це може викликати помилку компіляції. Розглянемо приклад. Нехай в заголовочному файлі `ratio.h` оголошено лише один структурний тип:

```
typedef struct tagRatio {
  int m, n;
} TRatio;
```

Нехай є ще два заголовочні файли, в яких оголошуються функції для роботи з об'єктами структурного типу `TRatio`. У файлі `ratio_io.h` оголосимо функції для введення та виведення цих об'єктів, а у файлі `ratio_m.h` — функції для математичних обчислень з ними. В кожному з цих двох заголовочних файлів першим рядком йде підключення файлу `ratio.h`:

Файл `ratio_io.h`:

```
#include "ratio.h"
TRatio ratio_read();
void ratio_print( TRatio );
```

Файл `ratio_m.h`:

```
#include "ratio.h"
TRatio ratio_add( TRatio, TRatio );
TRatio ratio_mpy( TRatio, TRatio );
```

Нарешті, нехай є модуль, який має намір використовувати обидва набори функцій: введення-виведення та математичні:

```
#include "ratio_io.h"
#include "ratio_m.h"
/* якийсь програмний текст */
```

Розберемо, як транслятор прочитає цей текст. Натрапивши на перший рядок, він (спрощено кажучи) вставить на це місце файл `ratio_io.h` та спробує обробити його, а у цьому файлі є директива включення файлу `ratio.h`, тому транслятор підставить вміст і цього файлу. Так само, обробляючи другий рядок, транслятор змушений буде ще раз вставити текст файлу `ratio.h`. Отже, з точки зору транстятора модуль після обробки всіх директив виглядатиме так:

```
typedef struct tagRatio {
  int m, n;
} TRatio;
TRatio ratio_read();
void ratio_print( TRatio );
typedef struct tagRatio {
  int m, n;
} TRatio;
TRatio ratio_add( TRatio, TRatio );
TRatio ratio_mpy( TRatio, TRatio );
/* якийсь програмний текст */
```

Як видно, при компіляції модуля транслятор побачить два означення структурного типу `TRatio` — а це є помилкою з точки зору граматики мови C.

Отже, як видно з цього прикладу, помилки можуть виникати через те, що один заголовочний файл підключається до кількох інших заголовочних файлів. Коли ці останні підключаються до якогось модуля, весь текст першого файлу включається багаторазово. Тому виникає задача: винайти такий спосіб написання та використання заголовочних файлів, який би гарантовано уберігав програміста від проблем з подвійним включенням. Іншими словами, хотілося б знайти такий засіб, який би примушував компілятор автоматично слідкувати за спробами багатократно підключити один заголовочний файл, щоб цей файл підключався лише один раз, при першій такій спробі. Саме такий засіб описано в наступному розділі.

12.4. Умовна компіляція

Мова C містить директиви препроцесора **`#ifdef`** та **`#ifndef`**, які дозволяють на етапі компіляції в залежності від тієї чи іншої умови вилучити або, навпаки, включити той чи інший фрагмент тексту.

Конструкцію

```
#ifdef ім'я
/* якийсь програмний текст */
#endif
```

компілятор обробляє так: спочатку він перевіряє, *чи означене* дане `ім'я`, тобто чи був десь раніше при компіляції цього ж модуля означений макрос з таким іменем; якщо ні, то весь програмний текст до директиви **#endif** пропускається, не компілюється, а якщо так, то компілюється. Директива **#ifndef** працює навпаки: вилучає з компіляції текст до директиви **#endif** тоді, коли макрос з вказаним іменем означений.

Ці директиви, які називають *директивами умовної компіляції*, мають багато різних застосувань, які, однак, далеко виходять за рамки ознайомчого курсу для початківців. Зараз опишемо лише одну конструкцію, яка дозволяє вирішити проблему повторного включення.

Розглянемо файл `ratio.h` у такій редакції:

```
#ifndef _RATIO_H_
#define _RATIO_H_

typedef struct tagRatio {
  int m, n;
} TRatio;

#endif
```

Нехай він, як і раніше, підключається до модуля двічі через посередництво інших заголовочних файлів. При першому включенні компілятор, натрапивши на директиву `ifndef`, продовжить компіляцію, оскільки макрос з іменем `_RATIO_H_` раніше не означувався. В наступному ж рядку даний макрос стає означеним. Після цього при другому включенні того ж заголовочного файлу компілятор, знов обробляючи директиву `ifndef`, помітить, що макрос `_RATIO_H_` вже означено, отже пропустить весь текст заголовочного файлу. Таким чином, вдалося запобігти повтору раніше відкомпільованих оголошень.

Продемонстрована тут техніка є стандартною для мови C: якщо придивитися до фірмових заголовочних файлів, що входять, наприклад, до комплекту середовища програмування, в них можна побачити той же технічний прийом.

Додаток А

Поглиблений опис деяких елементів мови

Мова C та пов'язаний з нею особливий стиль — це настільки масштабне та складне явище і ціла культура програмування, що годі й сподіватися викласти її в повному обсязі студентам, які лише починають знайомство зі спеціальністю. Всюди в попередніх частинах посібника розглядався лише мінімально необхідний набір засобів мови, якого в принципі достатньо для програмування більш-менш складних задач.

В цьому розділі опишемо деякі (але далеко не всі) додаткові можливості мови C, які допоможуть зробити програмні тексти значно лаконічнішими та виразнішими.

А.1. Форматні рядки функції printf

В розділі 2.7 коротко розглянуто основний спосіб застосування функції `printf` — друк значень різних типів разом з фрагментами тексту. Можливості цієї функції набагато ширші і дозволяють також управляти кількістю знакомісць, що виділяється на екрані для друку значення, вирівнювання по правій чи лівій границі, виведення знаку «плюс» перед додатнім числом тощо.

Вище неодноразово зазначалося, що цілі числа можуть бути знаковими та беззнаковими. У першому випадку старший біт у двійковому зображенні числа інтерпретується як знак (0 — для невід'ємного, 1 — для від'ємного числа), а в другому випадку він є ще одним значущим розрядом. Для друку цілого числа як беззнакового треба застосовувати специфікатор `%u`. Візьмемо, наприклад, число 56713. В його двійковому зображенні старший розряд має значення 1, і якщо це число надрукувати за допомогою специфікатору `%d`, старший біт буде сприйнято як знак «мінус», а специфікатор `%u` надрукує правильно:

```
unsigned int m = 56713;
printf( "неправильно␣%d,␣правильно␣%u\n", m, m );
```

Можна також керувати способом друку знаку перед числом. Звичайна поведінка функції виведення полягає в тому, що перед від'ємним числом друкується знак «мінус», а перед невід'ємним не друкується нічого. Однак, коли набір значень друкується у стовпчик, буває зручно перед додатнім числом, для наочності, ставити знак «плюс» або пробіл — порівняйте такі три стовпчики чисел:

```
21949    +21949     21949
-1611    -1611     -1611
-8380    -8380     -8380
2143     +2143      2143
```

Для такого способу друку, як показано у другому та третьому стовпчиках, у специфікаторі одразу після знаку `%` треба вказати прапорець — відповідно знак «плюс» або пробіл, наприклад цей фрагмент надрукує числа на зразок другого стовпчика:

```
#define N 4
int m[N] = { 21949, -1611, -8380, 2143 };
int i;
for( i = 0; i < N; ++i )
  printf( "%+d\n", m[i] );
```

Крім того, часто (особливо коли друкуються табличні дані) буває бажано задавати під друк значення певну ширину, тобто визначену кількість екранних знакомісць. Ширина вказується в специфікаторі формату, як в наступному прикладі:

```
#define N 4
int m[N] = { 21949, -1611, -8380, 2143 };
int i;
for( i = 0; i < N; ++i )
  printf( "%8d\n", m[i] );
```

Тут кожне число з масиву m друкується так, щоб зайняти рівно 8 екранних позицій. Якщо число має меншу кількість цифр, воно доповнюється зліва потрібною кількістю пробілів, тобто вирівнюється по правій границі поля. Отже, наведений програмний фрагмент надрукує такий результат:

```
␣␣␣21949
␣␣␣-1611
␣␣␣-8380
␣␣␣␣2143
```

Можна також доповнювати числа зліва не пробілами, а нулями — для цього ширина в специфікаторі повинна починатися з цифри 0: `printf( "%08d\n", m[i] )`. Нарешті, якщо перед шириною поставити знак «мінус», тобто якщо написати `printf( "%-8d\n", m[i] )`, то надруковані значення будуть вирівнюватися не по правому, а по лівому краю, тобто доповнюватися пробілами будуть справа.

Коли друкується дійсне число, буває корисно керувати не лише загальною шириною його друку, але й кількістю знаків, що відводяться під друк дробової частини, тобто точністю. Точність вказується у специфікаторі формату після ширини через крапку. Наступний фрагмент друкує одне й те саме значення тричі з різною точністю:

```
double pi = 3.1415926;
printf( "%9.2lf\n", pi );
printf( "%9.4lf\n", pi );
printf( "%9.6lf\n", pi );
```

Якщо замість числового значення ширини або точності вказати символ `*`, це означає, що ширину треба взяти з цілочисельного аргумента. Наступний фрагмент друкує число π в циклі зі зростанням точності, на кожній ітерації точність дорівнює поточному значенню змінної `i`.

```
double pi = 3.1415926;
int i;
for( i = 0; i < 8; ++i )
  printf( "%9.*lf\n", i, pi );
```

Ціле число можна надрукувати у вісімковій та 16-ковій системах числення. Для цього слугують специфікатори формату `%o` та `%x` або `%X`. Наприклад, наступний фрагмент надрукує значення числа в трьох системах числення:

```
int m = 1209;
printf("У␣10-ковій␣%d,␣8-ковій␣%o,␣16-ковій␣%X",
   m, m, m );
```

Якщо підсумувати все сказане, то в загальному випадку специфікатор складається з таких частин:

1. Символ `%`.
2. Прапорці `+`, `-`, пробіл;
3. Ширина (ціле число, що може починатися з цифри 0, або `*`);
4. Символ крапки та точність (ціле число або `*`);
5. Модифікатор довжини даних `l`;
6. Символ конвертації типу — найголовніший компонент специфікатору, присутній обов’язково, див. табл. А.1.

Табл. А.1. Символи конвертації типу у функції `printf`

Літера	Тип аргументу	Спосіб виведення
d	Ціле	Десяткова система, зі знаком
i	Ціле	Десяткова система, зі знаком
o	Ціле	Вісімкова система, без знаку
u	Ціле	Десяткова система, без знаку
x	Ціле	16-кова система, без знаку
X	Ціле	16-кова система, без знаку
f	Дійсне	Старші розряди друкуються
e	Дійсне	Експоненційна форма
c	Символ	
s	Рядок	

А.2. Присвоювання

Операція присвоювання, яка в загальному випадку має вигляд `x=E` (де `x` — ім'я змінної, а `E` — вираз), сама по собі є виразом, значенням якого є присвоєне значення, та може використовуватися у складі інших виразів.

Наприклад, розглянемо фрагмент програми:

```
int u = 6, v = 4, a, b, c;
c = (a = u+v) * (b = u-v);
```

Спочатку обчислюються підвирази `(a=u+v)` та `(b=u-v)`. Обчислення першого присвоює змінній `a` значення 10, і це ж число стає значенням виразу. Таким же чином, другий вираз дає значення 2 і, крім того, присвоює це число змінній `b`. Нарешті, змінній `c` присвоюється добуток отриманих двох значень. Наведений вище оператор повністю аналогічний наступним трьом:

```
int u = 6, v = 4, a, b, c;
a = u+v;
b = u-v;
c = a * b;
```

В розділі 3.1 було запроваджено операції інкременту `++` та декременту `--`. Всюди вище вони використовувалися лише в найпростіший спосіб, коли оператор цілком складається з операції інкременту або декременту, наприклад

```
x--;
++k;
```

Вирази вигляду `++x` чи `x++` можуть використовуватися у складі інших виразів, наприклад

```
int a = 3, b = 7, c;
c = (++a) * (b--);
```

Саме тут проявляється різниця між префіксною та постфіксною формами. Коли при обчисленні виразу в ньому трапляється префіксна операція, то спочатку збільшується або зменшується значення змінної і далі значенням операції `++x` (`--x`) стає нове значення цієї змінної. Натомість результатом постфіксної операції є старе значення змінної.

Розглянемо оператор

```
c = (++a) * (b--);
```

при початкових значеннях змінних: $a = 3$, $b = 7$. Операція префіксного інкременту збільшить на одиницю значення змінної `a`, і значенням виразу `(++a)` стане число 4. Операція декременту зменшує значення змінної `b`, але, оскільки операція постфіксна, значенням виразу `(b--)` є ще старе значення змінної `b`, тобто число 7. Отже, результатом цього оператору є такі значення змінних: $a = 4$, $b = 6$, $c = 4 \cdot 7 = 28$.

Для порівняння, при тих же початкових значеннях $a = 3$, $b = 7$ розглянемо оператор

```
c = (a++) * (--b);
```

Значення змінної `a` збільшується на 1, але в операції множення бере участь ще старе значення. Натомість другим операндом множення буде значення змінної `b` після зменшення. Отже, результат оператору такий: $a = 4$, $b = 6$, $c = 3 \cdot 6 = 18$.

А.3. Побітові операції

Крім загальновідомих операцій, таких як додавання та множення, мова C містить кілька цікавих операцій над цілими числами, що можуть бути дуже корисними для програмування складних алгоритмів обробки даних (особливо для шифрування, стискання, поміхостійкого кодування тощо).

Операції побітової кон'юнкції `&` та диз'юкції `|` (не плутати з логічними кон'юнкцією `&&` та диз'юкцією `||`!) виконуються біт за бітом над двійковими зображеннями двох чисел. Розглянемо, наприклад, оператор

```
int a = 177, b = 43, c;
c = a | b;
```

Переведемо числа 177 та 43 у двійкову систему числення, отримаємо відповідно 10110001 та 00101011. Підпишемо їх одне під одним та обчислимо по черзі кожен розряд результату. Результат диз'юнкції має в i-му розряді 1 тоді і тільки тоді, коли хоча б один з операндів має в i-му розряді 1.

$$\begin{array}{r|cccccccc} & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ | & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ \hline & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \end{array}$$

Після переходу до десяткової системи маємо результат 187.

Операція `^` побітової *виключної диз'юнкції* працює так: результат операції має в i-му розряді 1 тоді і тільки тоді, коли рівно один з операндів (але не обидва одразу) має в i-му розряді 1. Наприклад, обчислимо `177^43`:

$$\begin{array}{rcccccccc} & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ \wedge & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ \hline & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{array}$$

Після переходу до десяткової системи маємо результат 154.

Операція побітового заперечення позначається `~` і має один операнд. Результат має в i-му розряді 1 тоді і тільки тоді, коли операнд має в цьому розряді 0.

Нарешті, згадаємо операції побітового зсуву вліво `<<` та вправо `>>`. Значенням виразу `x<<y` є двійкове зображення значення `x`, зсунуте вліво на `y` бітів; біти, що звільняються, заповнюються нулями.

А.4. Умовна операція

Операція `?:` часто дозволяє замінити умовний оператор і зробити програмний код компактнішим, зекономивши два-три рядки. Вираз вигляду `e1?e2:e3` обчислюється так. Спочатку обчислюється значення виразу `e1`. Якщо це логічна істина, то кінцевим результатом стає результат обчислення виразу `e2`, а якщо хиба — то виразу `e3`.

Наприклад, оператор

```
m = (a > b) ? a : b;
```

присвоює змінній `m` найбільше зі значень змінних `a` та `b`. Порівняємо з умовним оператором, який робить те ж саме:

```
if( a > b )
  m = a;
else
  m = b;
```

Отже, якщо в залежності від істинності чи хибності умови треба виконати ту чи іншу дію, слід застосовувати умовний оператор **if**, а якщо йдеться лише про присвоювання одного з двох можливих значень, то варто застосувати умовну операцію.

A.5. Оператор переходу

Кожен оператор може бути помічений міткою, в якості мітки використовуються ідентифікатори. Оператор **goto** `мітка;` призводить до того, що виконання програми негайно переходить до оператору, поміченого даною міткою. Наприклад, розглянемо фрагмент

```
int a, b;
scanf( "%d %d", &a, &b );
loop:
  if( a == b ) goto end;
  if( a > b ) a -= b;
  else b -= a;
goto loop;
end:
printf( "%d\n", a );
```

Легко здогадатися, що це алгоритм Евкліда для обчислення найбільшого спільного дільника двох чисел. Замість оператору циклу в ньому застосовано таку конструкцію: початок тіла циклу помічений міткою `loop:`, а наприкінці тіла стоїть оператор переходу **goto** `loop;`, який примушує програму повторно виконати тіло циклу. Для виходу з циклу по досягненню певної умови призначений оператор **goto** `end;`, який передає управління на оператор, що стоїть після тіла циклу.

Областю видимості мітки є тіло функції, в якій вона застосована. Іншими словами, оператор **goto** може передати управління на інший оператор лише в межах цієї ж функції.

В ранній період розвитку програмування оператор безумовного переходу застосовувався доволі широко, і його опис стояв у підручниках на перших сторінках. Однак зараз важливим елементом програмістської культури є саме невикористання цього оператору. Математично доведено (зокрема, теорема про регуляризацію В.М.Глушкова), що будь-яку програму, яку можна написати з використанням оператору **goto**, можна перетворити до еквівалентного вигляду, з якого цей оператор всюди вилучено. Всі випадки використання оператору безумовного переходу можна замінити операторами розгалуження, циклу або викликами підпрограм. Практика підтверджує, що структуровані програми (тобто написані з використанням перерахованих засобів) набагато легше створювати, розуміти та змінювати, ніж програми, написані з використанням **goto**.

A.6. Покажчики на функції

Це одна з надзвичайно потужних і водночас непростих можливостей мови C. Кожна функція, її машинний код, має десь зберігатися в машинній пам'яті. Це значить, що для будь-якої функції осмисленим є питання «за якою адресою вона розташована?».

Мова C дозволяє оголошувати змінні типу покажчика на функцію та присвоювати їм в якості значень адреси наявних у програмі функцій.

Покажчик в мові C завжди позначається зірочкою. При оголошенні змінної типу покажчика на функцію треба зірочку разом з іменем змінної взяти в круглі дужки — інакше компілятор вважатиме, що зірочка відноситься не до імені змінної, а до типу значення, що повертає функція. Наприклад:

```
int *f( int );
int (*p)( int );
int *(*q)( int );
```

В першому з цих рядків оголошено функцію `f`, яка має один цілий аргумент та повертає значення типу «покажчик на ціле». В другому рядку оголошено змінну `p`, тип якої «покажчик на функцію, яка має один цілий аргумент та повертає ціле число». Нарешті, в третьому рядку оголошено змінну `q` типу «покажчик на функцію, яка має один цілий аргумент та повертає покажчик на ціле».

Змінній типу покажчика на функцію можна присвоїти адресу будь-якої наявної в програмі функції, яка має таку ж кількість і типи аргументів і тип значення. Наприклад, якщо дано оголошення

```
int f1( int, int );
int f2( int, int );
void h1( int );
int h2( int );

int (*pf)( int, int );
```

то законними є присвоювання `pf=&f1` та `pf=&f2`, а присвоювання `pf=&h1` та `pf=&h2` — помилкові.

Основна операція над змінною типу покажчика на функцію — це виклик. В наступному операторі викликається та функція, на яку в даний момент вказує змінна `pf` (а це може бути одна з двох функцій, `f1` або `f2`):

```
x = pf( 8, -1 );
```

Особливо широкі можливості відкриває використання покажчиків на функції в якості аргументів інших функцій. Розглянемо функцію

```
void apply( int *p, int n, void (*h)(int) ) {
  int i;
  for( i = 0; i < n; i++ )
    h( p[i] );
}
```

Перші два аргументи цієї функції — це покажчик на масив цілих чисел та кількість елементів в ньому. Третій аргумент — покажчик на деяку функцію, яка має один цілочисельний аргумент. Як видно з тіла функції `apply`, вона по черзі застосовує функцію-аргумент до кожного елементу масиву. Продовжимо приклад:

```
void print_int_line( int x ) {
  printf( "%d\n", x );
}
void print_int_tab( int x ) {
  printf( "%d\t", x );
}

#define N 5

int main() {
  int m[N] = {18, 209, -1, 43, 116};
  apply( m, N, &print_int_line );
  apply( m, N, &print_int_tab );
  return 0;
}
```

Функція `print_int_line` друкує значення свого аргументу в окремому рядку, а функція `print_int_tab` друкує аргумент та робить табуляцію. Перший виклик функції `apply` застосовує функцію `print_int_line` до кожного по черзі елементу масиву `m`, а це означає, що кожен

елемент масиву друкується в окремому рядку. Другий виклик таким же чином друкує кожен елемент масиву через табуляцію.

Отже, використання покажчиків на функції дозволяє зробити програму більш гнучкою та універсальною.

А.7. Статичні змінні у функціях

В розділі 5.3 було викладено, як поводять себе локальні змінні функції: вони створюються при виклику цієї функції та знищуються при її завершенні. Тобто час життя локальної змінної співпадає з часом виконання функції, у якій вона оголошена. Таким чином, якщо деяка функція викликається кілька разів, то її локальні змінні щоразу знищуються та створюються заново.

Мова C містить цікаву можливість оголошувати у функціях також *статичні* змінні, у яких областю видимості, як і в звичайних локальних змінних, є тіло функції, але часом життя є весь час виконання програми. Статична змінна `x`, оголошена у функції `f`, у періоди між викликами функції `f` залишається у пам'яті, хоча і недоступна для будь-яких інших функцій. Коли програма наступного разу викличе функцію `f`, функція знов «побачить» те значення змінної `x`, яке було в неї при попередньому виклику цієї функції.

Розберемо приклад. У наведеній нижче функції `maxOf2`, яка знаходить більше з двох чисел, статична змінна `count` спершу отримує початкове значення 0. Треба звернути увагу: на відміну від оператору присвоювання, ініціалізація (присвоювання початкового значення при оголошенні змінної) виконується лише тоді, коли змінна створюється, а змінна в даному випадку статична і створюється лише один раз.

```
int maxOf2( int a, int b ) {
  static count = 0;
  ++count;
  printf( "maxOf2:␣%d-й␣виклик\n", count );
  return (a > b) ? a : b;
}
```

При кожному виклику цієї функції значення змінної `count` нарощується на одиницю (і запам'ятовується до наступного виклику), а на екран друкується повідомлення про кількість викликів. Таким чином можна вести статистику використання функцій.

Додаток Б

Небезпечні помилки

В попередніх розділах неодноразово відзначалося, що мова C нелегка для початківця ще й тому, що в ній багато «підводних каменів» — таких конструкцій, сенс яких змінюється до невпізнанності від помилки в одному-єдиному символі. Помилку з одного символа легко зробити з неуважності та дуже важко помітити в тексті програми.

При пошуку багатьох помилок величезну допомогу програмісту може надати компілятор. Наприклад, якщо в імені функції `printf` замість літери «`p`» набрати літеру «`q`», компілятор одразу повідомить, що в програмі робиться спроба викликати невідому функцію `qrintf` та вкаже на відповідне місце в тексті програми. Натомість особливо небезпечні помилки, про які йдеться тут, небезпечні тим, що синтаксично правильний текст при заміні чи пропуску одного символу перетворюється на інший текст, правильний з точки зору правил граматики мови, але вже з іншим сенсом. Компілятор не вміє читати думок і тому прочитає в тексті програми те, що написано, а не те, що програміст мав на увазі.

Крім того, багато типових помилок виникає через неповне розуміння того, як працюють ті чи інші конструкції мови C. Особливо це стосується наявних перетворень типів, покажчиків та засобів управління динамічною пам'яттю.

В цьому додатку зібрано каталог таких типових та дуже неприємних помилок. Кожен початківець повинен добре знати їх, щоб не марнувати години, ламаючи голову над тим, як абсолютно правильна на вигляд програма видає нісенітницю замість результатів.

Б.1. Ділення цілих та дійсних чисел

Початківці часто забувають, что ціле число та рівне йому за величиною дійсне число є з точки зору мови C різними об'єктами. Розглянемо приклад програми, яка, за задумом автора, повинна була б обчислювати суму

$$S = \sum_{i=1}^{n} \frac{1}{i}.$$

```
double s = 0;
int i, n;
scanf( "%d", &n );
for( i = 1; i <= n; ++i )
  s += 1 / i;
printf( "результат␣%lf\n", s );
```

Ця програма для будь-якого $n > 0$ надрукує результат 1. Справа в тому, що у виразі `1/i` обидва операнди операції ділення належать до цілого типу, і тому ділення виконується націло, з відкиданням дробової частини, щоб результатом ділення стало також ціле число. На першій ітерації циклу, при $i = 1$, ділення дає результат 1, а на всіх наступних ітераціях $i > 1$, тому ділення дає 0.

Щоб програма запрацювала правильно, треба привести ціле число до дійсного типу — якщо хоча б один аргумент операції ділення є дійсним числом, то і результатом також буде дійсне:

```
s += 1 / (double) i;
```

Б.2. Пропущений знак = в логічному виразі

Розглянемо два умовні оператори:

```
int x, y;
if( x == y ) {/* якісь оператори */}
if( x = y ) {/* якісь оператори */}
```

Людині на перший погляд важко помітити різницю між ними, однак сенс цих операторів та результати їх виконання відрізняються докорінно.

У першому випадку в дужках стоїть операція порівняння `==`. Цей оператор працює так: спочатку беруться значення змінних `x` та `y` та порівнюються. Операція порівняння дає результат цілого типу: число 1, якщо значення однакові, та число 0, якщо вони різні. Потім оператор **`if`** бере це цілочисельне значення та виконує `оператори` тільки тоді, коли воно не дорівнює 0. Отже, `оператори` виконаються тільки тоді, коли значення змінних `x` та `y` однакові.

В другому випадку в дужках стоїть не порівняння, а присвоювання. Операція бере значення змінної `y`, присвоює його змінній `x`. Значенням усього виразу `x=y` стає це присвоєне значення. Саме воно поступає до оператору **`if`** та визначає, чи будуть виконуватися `оператори`. Іншими словами, в другому випадку оператор насправді працює так:

```
x = y;
if( x != 0 ) {/* якісь оператори */}
```

Припустимо, що змінні `x` та `y` мають значення відповідно 3 і 4. Тоді в першому випадку `оператори` виконуватися не будуть (бо значенням виразу `3==4` є число 0), а в другому будуть (бо значенням виразу `y=4` є число 4). Припустимо тепер, що обидві змінні мають значення 0. Тоді в першому випадку умовний оператор спрацює (порівняння двох нулів дасть значення «істина», тобто число 1), а в другому не спрацює (значенням операції присвоювання буде присвоєне значення 0, тобто логічне значення «хиба»).

Чому ж компілятор не вважає другий випадок синтаксичною помилкою? Справа в тому, що автори мови C прагнули якнайбільшої універсальності. В ролі умови в умовному або циклічному операторах може стояти не лише вираз з операцією порівняння (`==`, `!=`, `>` тощо), а будь-який арифметичний вираз. З одного боку, це розширює коло можливих програм та робить їх тексти компактнішими, але з іншого — зворотнім боком такої гнучкості та універсальності стає небезпека необачного використання.

Б.3. Узгодженість специфікаторів формату

Зустрічаючи в програмі текстові константи, компілятор слідкує лише за їх синтаксичною правильністю і не цікавиться змістом рядка — за правильність значень констант відповідає програміст. Це особливо важливо пам'ятати при роботі з форматними рядками функцій `printf`, `scanf` та інших їм подібних. Зокрема, якщо кількість специфікаторів через неуважність програміста не збігається з кількістю решти аргументів, або якщо специфікатори не відповідають типам цих аргументів, поведінка програми може виявитися дивною або навіть непередбачуваною. Наприклад:

```
char str[80];
int m, a, b, c;
. . .
scanf( "%s␣%d", &m, str );
```

Легко помітити, що програміст всього лише переплутав місцями два аргументи, і тепер аргумент `&m` відноситься до специфікатора `%s`, а аргумент `str` — до специфікатора `%d`. Коли користувач введе текстовий рядок (що може складатися з десятків символів), функція спробує розмістити його в пам'яті в комірки, починаючи з тієї, де лежить змінна `&m` і при цьому, звичайно ж, зіпсує змінні `a`, `b` та `c`.

Б.4. Відсутність & в аргументі функції scanf

Аргументи функції `scanf`, що йдуть після форматного рядка, повинні бути *покажчиками* на ті змінні, в які потрібно розмістити значення, введені з клавіатури. Наприклад, щоб ввести з клавіатури значення в дві змінні цілого типу, потрібно написати так:

```
int a, b;
scanf( "%d%d", &a, &b );
```

Амперсанди перед іменами змінних обов'язково потрібні, щоб функції передавалися саме адреси змінних. Дуже часто недосвідчені програмісти припускаються помилки, не ставлячи амперсанди. Цей різновид помилок відноситься до особливо небезпечних, оскільки вони не «ловляться» компілятором. Розберемо, що відбудеться, якщо програміст напише

```
scanf( "%d%d", a, b );
```

На відміну від функцій, що мають заздалегідь визначений перелік аргументів, кожен з яких має цілком однозначно визначений тип, функція `scanf` має невизначену, довільну кількість аргументів (справді, її можна використовувати для введення однієї, двох, п'ятьох змінних). Правильність типів аргументів, що передаються таким функціям, компілятор не може перевірити автоматично. Якщо функція в мові C має невизначену кількість аргументів, то кожен з них може мати будь-який тип.

Отже, в прикладі, що розглядається, буде взято значення змінних `a` та `b` (нехай це будуть, наприклад, числа 4158 та 518) і передано у функцію `scanf`. Функція, однак, очікує, що передані їй аргументи будуть покажчиками, тому вона розгляне ці числові значення так, ніби вони були б не цілими числами, а адресами якихось комірок. Звичайно ж, це будуть безглузді адреси, тобто не адреси змінних, які належать цій програмі, а можливо адреси даних іншої програми, адреси кодів машиних команд або адреса життєво важливих даних операційної системи. Коли функція `scanf` прочитає з клавіатури два числа, вона запише їх в пам'ять за цими безглуздими адресами, що може зіпсувати дані чи код іншої програми або навіть викликати крах операційної системи.

Отже, вводячи значення змінних за допомогою функції `scanf`, треба уважно слідкувати, щоб до функції `scanf` передавати не значення, а адреси змінних.

Чи бувають випадки, коли справді потрібно застосовувати в аргументах функції `scanf` імена змінних без амперсандів? Звичайно ж, це має сенс, якщо змінна сама має тип покажчика та вказує на іншу змінну, в яку потрібно покласти введене значення, див. приклад:

```
double x, *p;
p = &x;
scanf( "%lf", p );
```

Б.5. Числові значення символів-цифр

В мові C тип **char** виконує одночасно дві ролі: це і цілочисельний тип з довжиною розрядної сітки 8 біт, і спеціальний тип для обробки символів. Кожну символьну константу, побачену в тексті програми, компілятор перетворює на ціле число — код відповідного символу — і надалі «не помічає» різниці між власне числами та числами–кодами символів.

Розглянемо фрагмент програми

```
char c1, c2;
c1 = 9;
c2 = '9';
```

Константа в правій частині другого оператору присвоювання відрізняється в тексті лише наявністю одинарних лапок. Але сенс цих операторів зовсім різний. Перший оператор присвоює змінній `c1` значення 9, а другий робить значенням змінної `c2` число, яке є кодом символу «9» — за таблицею ASCII це число 57.

Отже, програмісту потрібно мати на увазі різницю між цифровим символом та числом, яке ця цифра означає, та не забувати позначати символьні константи з одинарними лапками.

Б.6. Символьні константи та однолітерні рядки

Розглянемо константи `'a'` та `"a"`. Різниця в написанні лише та, що перша константа взята в одинарні, а друга — в подвійні лапки, і недосвідчений програміст може їх легко переплутати. Разом з тим, сенс цих констант зовсім різний.

В першому випадку маємо символьну константу, тобто ціле число, що є кодом літери «а». В другому випадку показана рядкова константа. Як роз'яснювалося в розділі 8, рядок в мові C розглядається як покажчик на масив символів, наприкінці якого стоїть спеціальний символ з числовим кодом 0. Якщо десь у тексті програми є константа `"a"`, то компілятор розглядає її як покажчик на масив з двох елементів, перший елемент якого це числовий код літери «а», а другий — число 0.

Отже, символьна константа та рядкова константа, яка складається з одного символу — це зовсім різні об'єкти, і плутати їх між собою неприпустимо.

Б.7. Порівняння масивів та адрес

Масиви в мові C не становлять повноцінного *типу* даних (хоча є, безперечно, *структурами* даних). Для масиву як цілого не мають сенсу операції порівняння. Масиви в мові C розглядаються через покажчики на перший елемент, отже операція порівняння порівнює, чи вказують два покажчики на одну й ту саму адресу. Розглянемо приклад:

```
int k[3] = {17, 93, 48};
int t[3] = {17, 93, 48};
if( k == t )
  printf( "однакові" );
else
  printf( "різні" );
```

Недосвідчений програміст міг би очікувати, що ця програма видасть повідомлення, що масиви однакові. Однак масиви `k` і `t`, хоча й складаються з однакових чисел, розташовані в двох різних областях пам'яті. Імена `k` і `t` іменують не сукупності елементів, а адреси цих областей. Тому операція порівняння порівняє адреси та дасть результат «хиба».

Для того, щоб порівняти вміст масивів замість їх адрес, програмісту потрібно описати алгоритм поелементного порівняння. Такий алгоритм можна було б розмістити в тому ж місці програми, безпосередньо перед умовним оператором, але загальноприйнята практика розробки програм та «дух» структурного програмування вимагають, щоб він був винесений в окрему функцію, наприклад:

```
int equalIntArrays( int *p, int *q, int n ) {
  int i;
  for( i = 0; i < n; i++ )
    if( p[i] != q[i] )
      return 0;
  return 1;
}
```

Ця функція продивляється два масиви елемент за елементом. Якщо хоча б в одній парі елементів виявиться відмінність, функція повертає значення «хиба»: масиви відрізняються між собою. Якщо ж до кінця масиву жодної розбіжності не виявлено, функція повертає значення «істина». Тоді попередній приклад програмного фрагменту перетворюється на такий:

```
int k[3] = {17, 93, 48};
int t[3] = {17, 93, 48};
if( equalIntArrays(k, t, 3) )
```

```
    printf( "однакові" );
  else
    printf( "різні" );
```

Фактично, стандартна функція strcmp для порівняння рядків (масивів символів) працює подібним чином (з тією лише відмінністю, що довжина масиву не задається окремим аргументом, а визначається за нуль-символом).

Операція порівняння == справді має сенс тоді, коли програмісту треба перевірити, чи вказують два покажчики на одну й ту саму область пам'яті. Іншими словами, коли треба перевірити не те, чи складаються два масиви з однакових елементів, а те, чи є вони одним і тим самим об'єктом.

Б.8. Присвоювання масивів та адрес

Так само, як і розглянута вище операція порівняння, операція присвоюваня не має сенсу для масивів. Розглянемо приклад:

```
int m[3] = {4, 63, -17};
int q[3];
q = m;
```

Недосвідчені програмісти іноді пишуть конструкції, подібні до наведеної, сподіваючись, що операція присвоювання скопіює всі елементі з масиву m в масив q. Насправді ж оператор присвоювання q=m неприпустимий з точки зору правил мови C.

Ім'я масиву в мові C іменує не сукупність елементів, а покажчик на область пам'яті, де масив розташовано. До того ж, це не просто покажчик, а константа-покажчик. Тобто, якщо змінній типу покажчика можна присвоїти нове значення, щоб вона вказувала на іншу адресу в пам'яті, то імена масивів, в даному прикладі m та q, жорстко зв'язані з відповідними областями пам'яті, і «примусити» їх вказувати на інші адреси в процесі виконання програми в принципі неможливо. Тому в наведеному вище прикладі буде помилка компіляції.

Натомість, для того щоб присвоїти значення з кожного елементу одного масиву до відповідного елементу іншого масиву, потрібно застосувати цикл, який за кожну ітерацію присвоює по одному елементу. За загальноприйнятою практикою будь-яку відносно самостійну задачу потрібно вирішувати за допомогою спеціальної функції:

```
void copyIntArray( int *to, int *from, int n ) {
  int i;
  for( i = 0; i < n; i++ )
    to[i] = from[i];
}
```

Першим аргументом функції є покажчик на початок того масиву, до якого треба копіювати значення, другим аргументом — покажчик на початок масиву, з якого потрібно взяти ці значення, а третій аргумент — це число елементів, які треба скопіювати, тобто довжина масивів. Тоді наведений вище помилковий програмний фрагмент набуває такого правильного вигляду:

```
int m[3] = {4, 63, -17};
int q[3];
copyIntArray(q, m, 3);
```

Таким же чином влаштована і стандартна функція strcpy для копіювання рядків.

Наостанок треба зазначити, що операція присвоювання, звичайно ж, має сенс для покажчиків на масиви, пам'ять для яких було виділено динамічно. Справді, робота з динамічними масивами здійснюється через змінну типу покажчика, а змінній за означенням можна присвоювати нові значення. Тому наведений нижче програмний фрагмент скомпілюється та виконається:

```
int *m, *q;
m = (int*) malloc(3 * sizeof(int));
m[0] = 4;
m[1] = 63;
m[2] = -17;
q = (int*) malloc(3 * sizeof(int));
q = m;
```

Але, якщо добре уявляти собі сенс та механізми роботи операторів, легко зрозуміти, що оператор присвоювання зробить зовсім не копіювання значень з масиву `m` до масиву `q`. Натомість він просто встановить покажчик `q` на той же масив, на який вказує покажчик `m`: два покажчики будуть вказувати не на два однакових, але окремих масиви, а на один і той самий масив. Крім того, в наведеному прикладі є ще одна широко розповсюджена та підступна помилка, так званий *витік пам'яті*, про який буде йти мова далі в цьому розділі.

Б.9. Крапка з комою в структурному операторі

Розглянемо програму, яка, за задумом програміста, мала б друкувати числа від 1 до 10, але в якій програміст поставив зайвий знак «крапка з комою»:

```
for( i = 1; i <= 10; i++ );
  printf( "%d\n", i );
```

Слово **for** зв'язує лише *один* оператор, що стоїть після нього. В мові С крапка з комою сама по собі теж є своєрідним *порожнім* оператором, який означає «*нічого не робити*». Тому тілом циклу є порожній оператор, а виклик функції `printf` не має жодного відношення до циклу — транслятор побачить, що цей виклик просто стоїть *після* циклу. Отже, програма 10 разів підряд зробить *нічого*, а лише після того, як значення лічильника `i` досягне 11 та цикл розірветься, один раз буде надруковано значення лічильника.

Все сказане стосується також ключових слів **if**, **else**, **while**. Наприклад, якщо зайву крапку з комою поставити в умовному операторі

```
if( x > 0 );
  printf( "a\n" );
```

то транслятор так зрозуміє семантику програми: якщо значення змінної `x` додатне, то виконати порожній оператор та перейти до наступного оператору, в протилежному випадку — перейти до наступного оператору. Тобто в обох випадках, незалежно від істинності умови, буде виконано виклик функції `pprintf`, оскільки той не має жодного відношення до умовного оператору.

Підкреслимо, що наведені вище програмні фрагменти скомпілюються нормально, але будуть виконувати не ті дії, які мав на увазі програміст. В наступному прикладі поставимо зайву крапку з комою в умовному операторі, який має другу гілку **else**.

```
if( x > 0 );
  printf( "a\n" );
else
  printf( "b\n" );
```

Ця програма, на відміну від попередніх, не скомпілюється, транслятор зафіксує синтаксичну помилку. Справді, ключове слово **else** може стояти лише одразу після першої гілки умовного оператору. Але ж гілкою умовного оператору є порожній оператор, побачивши після нього виклик функції `prntf`, транслятор зрозуміє, що умовний оператор закінчився та не містить другої гілки. Тепер слово **else** «висить в повітрі», тобто не пов'язане з жодним словом **if**.

Заради точності зауважимо, що іноді професіонали високого класу навмисно використовують порожній оператор «(;)» як тіло циклу. Але такі «фігури вищого пілотажу» вимагають

досвіду і навичок і чіткого усвідомлення, що саме програміст хоче «сказати» машині, який задум програми і як цей оператор працює. Розглянемо один приклад:

```
double p = 1;
int n, i;
scanf( "%d", &n );
for( i=n; i; p *= i-- );
printf( "%lf\n", p );
```

Ця програма обчислює та виводить на екран факторіал введеного користувачем числа `n`. Факторіал обчислюється у змінній `p`, її початкове значення дорівнює 1, змінна `i` пробігає по черзі значення від `n` до 1 у порядку спадання.

Звернімо увагу на третій вираз в операторі **for**. Згідно семантики мови C, цей вираз обчислюється наприкінці кожної ітерації. Прийом вищого пілотажу тут полягає в тому, що даний вираз містить одразу дві операції присвоювання: спочатку поточне значення змінної `p` домножається на значення змінної `i`, а результат присвоюється знов у змінну `p`, потім значення змінної `i` зменшується на одиницю. Іншими словами, тут в один вираз об'єднано і власне зміну параметру циклу (зменшення змінної `i` на одиницю), і обчислення бажаного результату, яке зазвичай виносять у тіло циклу.

Умова завершення, яка перевіряється перед кожною ітерацією, складається з самої лише змінної `i`. На черговій ітерації, коли значенням змінної `i` буде число 1 і змінна `p` домножиться на одиницю, після операції декременту значенням змінної `i` стане 0, одразу після цього буде перевірено умову продовження циклу, а число 0 є логічною хибою.

Б.10. Складні умови

Нехай потрібно написати програму, яка розпізнає, чи належить введене користувачем число x до інтервалу $*(-2, +2)$, тобто перевіряє умову $-2 < x < 2$. Ось дуже типовий помилковий розв'язок:

```
double x;
scanf( "%lf", &x );
if( -2 < x < 2 )
  printf( "належить" );
else
  printf( "не␣належить" );
```

Як не дивно, цей розв'язок буде відповідати «належить» на будь-яке значення x.

По-перше, логічні значення «істина» та «хиба» кодуються в мові C цілими числами. Результатом операції порівняння `<` або `>` стає число 0 або 1. По-друге, операції порівняння, як і більшість інших, групуються зліва направо, тобто компілятор обчислює вираз `-2<x<2` як `(-2<x)<2`. Отже, наведений вище фрагмент виконується так: спочатку обчислюється значення виразу `-2<x`, і його значенням стає або число 0, або число 1. Потім цей результат порівнюється з числом 2 і завжди виявляється меншим.

Щоб складна умова працювала правильно, треба збирати її з більш простих за допомогою логічних операцій кон'юнкції та диз'юнкції. В даному прикладі слід було б написати так:

```
if( (-2 < x) && (x < 2) )
```

Б.11. Логічна та побітова кон'юнкція і диз'юнкція

Логічна кон'юнкція позначається двома знаками амперсанда `&&`. В мові C є також операція, яка позначається одним знаком амперсанда `&`, це побітова кон'юнкція. Так само, логічна диз'юнкція позначається двома вертикальними рисками `||`, а побітова — однією `|`. Схожість позначень та певна близькість сенсу цих операцій можуть спричинити до плутанини.

Логічна кон'юнкція двох цілочисельних значень, нагадаємо, працює так: якщо обидва операнди не дорівнюють нулю (що слугує моделлю логічної істини), то результатом є число 1, в будь-якому іншому випадку, тобто коли хоча б один з операндів має значення 0 (хиба), результатом операції є число 0. Наприклад, змінна `f` отримає значення 1 тоді і тільки тоді, коли одночасно вірні обидві нерівності:

```
double p, q, r;
int f;
f = (p<q) && (q<r);
```

Побітова кон'юнкція натомість працює з кожним бітом операндів окремо. Обчислимо, наприклад, значення виразу `69~&~38`. Переведемо спочатку обидва числа в двійкову систему: $69_{10} = 01000101_2$, $38_{10} = 00100110_2$.

Тепер підпишемо ці числа у двійковому зображенні один під одним та будемо обчислювати кожен по черзі двійковий розряд результату: k-й розряд результату дорівнює 1 лише тоді, коли одиниці дорівнює k-й розряд в обох операндах. Маємо:

$$\begin{array}{cccccccc} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}$$

Нарешті, перетворимо результат у десяткову систему числення: $00000100_2 = 4_{10}$.

Отже, важливо розуміти, що логічна та побітова кон'юнкції (так само, диз'юнкції) — зовсім різні операції: логічна працює з числами як з суцільними об'єктами, а побітова розбиває числа на розряди. Зокрема, легко навести приклади таких пар чисел, що їх логічна кон'юнкція дорівнює 1, а побітова 0, такі хоча б числа 1 та 2, 4 та 16 тощо.

Логічна кон'юнкція та диз'юнкція використовується для того, щоб утворити більш складний логічний вираз або умову з простіших: наприклад, цей оператор перевіряє, чи лежить значення змінної `x` в проміжку від 0 до 1.

```
if( (x>0) && (x<1) ) ...
```

Натомість, побітова кон'юнкція та диз'юнкція — це арифметичні операції, специфічні для двійкової системи числення.

Б.12. Помилки при роботі з пам'яттю

Вище розглянуто семантичні помилки, які часто виникають через оманливу синтаксичну подібність різних за сенсом елементів мови C. Наостанок розглянемо помилки іншого роду, також дуже характерні для програмування мовою C та непрості для пошуку та виправлення.

Програмування мовою C — це майже завжди робота з покажчиками. Покажчики великою мірою визначають «дух» програмування на C. Але покажчики являють собою дуже специфічний тип даних. Цілі та дійсні числа добре знайомі кожній людині, людина сприймає їх майже інтуїтивно. Покажчики, натомість, не властиві постановці прикладної задачі як такій, це суто програмістський, машинно-орієнтований засіб, тому повинен пройти певний час, перш ніж програміст-початківець навчиться оперувати ними вправно.

Найрозповсюдженіша помилка — *неініціалізований*, або, як його ще називають, *дикий* покажчик. Будь-яка змінна, оголошена в програмі, до того моменту, як програма вперше присвоїть їй якесь значення, містить довільне, непередбачуване, безглузде значення. Якщо це змінна типу покажчика, то це означає, що вона містить адресу якої завгодно комірки пам'яті, тобто вказує невідомо куди. Будь-яка спроба працювати з таким покажчиком означає звертання за цією випадковою адресою. Наприклад:

```
int *p; /*оголошено, але значення не присвоєно*/
int x;
x = *p + 1; /* на які дані вказує p? безглуздо */
p[8] = -1; /* небезпечно! */
```

Покажчики можуть використовуватися в двох основних ролях: або як адреси «звичайних» змінних, або як покажчики на динамічно виділені області пам'яті. В другому випадку помилки, пов'язані з «дикими» покажчиками, особливо часто виникають через те, що програміст, забувши виділити пам'ять, починає працювати з покажчиком так, ніби він вже вказує на справжню область даних. В наведеному нижче прикладі переплутано місцями рядки: треба було спочатку виділити пам'ять, а потім вже заповнювати масив нулями.

```
int *p, n=10, i;
for( i = 0; i < n; i++ )
  p[i] = 0; /* p поки що вказує невідомо куди */
p = (int*) malloc( n * sizeof(int) );
```

Ще одне джерело диких покажчиків — спроба працювати з покажчиком на область динамічної пам'яті, яка була раніше виділена, але потім звільнена. Приклад:

```
int *p, n=10;
p = (int*) malloc( n * sizeof(int) );
p[0] = -1; /* нормально: масив вже створено */
free( p ); /* знищення масиву */
p[1] = 8; /* p вказує на неіснуючий масив! */
```

Після виклику функції `free` покажчик `p`, звичайно ж, не починає раптом вказувати на іншу адресу (це й неможливо: функція не може змінити змінну, значення якої передано їй в якості аргументу). Але область пам'яті, виділена раніше функцією `malloc` та на яку вказує покажчик, відтепер знов стає вільною, отже в будь-який момент часу машина може, скажімо, передати цю область іншій програмі, яка попросить надати їй пам'ять, викликавши функцію `malloc`. В будь-якому разі, не гарантується, що значення, присвоєне елементу масиву в останньому рядку прикладу, збережеться і не буде зіпсоване іншими присвоюваннями.

Нарешті, ще один характерний різновид диких покажчиків виникає при спробі повернути з функції покажчик на її локальну змінну. Нагадаємо, що час життя локальної змінної, оголошеної в певній функції, це в точності той час, поки ця функція виконується. Локальні змінні створюються в момент виклику функції та знищуються при поверненні з неї.

```
int *f() {
  int x = 0; /* локальна */
  return &x; /* повернути адресу локальної */
}

int main() {
  int *p;
  p = f(); /* 1 */
  *p = 0; /* 2 */
  . . .
  return 0;
}
```

Звернімо увагу на оператор функції `main`, помічений цифрою 1. Тут викликається функція `f`, а значення, яке вона повертає, присвоюється у змінну `p`. Але ж функція `f` повертає адресу своєї локальної змінної `x`. На момент повернення значення у функцію `main` ця змінна вже знищена — покажчик `p` показує лише на місце, де раніше була змінна. Звичайно ж, це вільне місце машина може в будь-який момент використати, розмістивши там іншу змінну. Отже, наведена програма безглузда.

Вище розглядалися помилки, коли покажчик є, а об'єкта, на який би він вказував, немає. Часто трапляється і протилежний випадок, коли динамічно створений об'єкт продовжує існувати в пам'яті, а покажчика на нього немає. Це так звані дикі об'єкти. Розглянемо кілька прикладів.

```
int *p, n=10;
p = (int*) malloc( n * sizeof(int) ); /* 1 */
```

```
p[0] = -1;
n = 20;
p = (int*) malloc( n * sizeof(int) ); /* 2 */
p[0] = 8;
```

Спочатку створюється динамічний масив, а його адреса присвоюється в змінну `p`. Ця змінна — єдиний ключ, за допомогою якого програма може звернутися до області пам'яті, виділеної функцією `malloc`. В рядку, відміченому цифрою 2, виділяється нова область динамічної пам'яті і покажчик на неї присвоюється у змінну `p`. При цьому старе значення змінної безповоротно втрачається. Отже, масив, створений у рядку з поміткою 1, так і залишається у пам'яті на тому ж місці і продовжує зберігати дані, але у програми немає жодної можливості до нього звернутися. Зрозуміло, щоб позбутися цієї неприємності, треба було б перед рядком з поміткою 2 звільнити пам'ять від попереднього масиву, викликавши `free(p)`.

В другому прикладі динамічний масив створюється у функції. Адреса масиву присвоюється локальній змінній цієї функції. Після повернення з функції локальна змінна знищується, але ж динамічний масив, на який вона вказує, залишається, і відтепер у програми немає можливості до цього масиву звернутися. У програми навіть немає можливості його знищити, бо для цього треба мати адресу об'єкту, щоб передати її як аргумент до функції `free`.

```
void f() {
  int *p, n=10;
  p = (int*) malloc( n * sizeof(int) );
  p[0] = 1;
}

int main() {
  f();
  . . .
  return 0;
}
```

На відміну від диких покажчиків, дикі об'єкти не становлять критичної небезпеки, вони не призводять до краху програми, до псування важливих даних. Такі об'єкти просто лежать в пам'яті без діла, займаючи простір та не даючи використати пам'ять для інших даних. Накопичення в пам'яті диких об'єктів також називають *витіканням пам'яті*.

Додаток В

Поради щодо стилю

Знання мови програмування самої по собі, її граматики, значення кожного слова чи знаку, безумовно, лежить у фундаменті професії програміста. Але не менш важливе вміння писати тексти програм не просто правильно, а ще й красиво.

Програмування не обмежується спілкуванням людини з машиною, текст програми призначений для сприйняття не лише комп'ютером. Сучасне програмування передбачає спільну діяльність доволі великих колективів над кожним програмним продуктом. В цих умовах на перший план виходить гарне оформлення програмного тексту, яке сприяє його розумінню сторонньою людиною.

Навіть коли над програмою працює лише одна людина, у автора виникає потреба неодноразово повертатися до раніше розробленого тексту, іноді через кілька місяців. Якщо текст програми оформлено недбало, без дотримання певної дисципліни, то розібратися у своєму ж тексті може виявитися нелегкою та довгою справою. Може навіть статися, що написати той чи інший фрагмент програми заново буде легше, ніж внести в наявний текст невелкі корективи.

Правила гарного стилю сильно відрізняються за своєю природою та сенсом від правил граматики мови C. По-перше, відступ від правил граматики одразу ж фіксується транслятором, а відступ від правил гарного стилю ні. Транслятору байдуже до наявності чи відсутності в тексті пробілів, порожніх рядків, коментарів. Недотримання цих правл відчутне лише для людини. По-друге, правила граматики носять недвозначний та чіткий характер, а правила стилю визначаються інтуїтивними уявненнями людини про зручність та красу, які не можуть бути однозначно формалізовані. Якщо граматика мови єдина для всіх програмістів, то деякі особливості стилю кожен виробляє для себе особисто.

Наведені нижче правила не є обов'язковими для виконання. Скоріше, на їх основі студентам пропонується винайти свій власний стиль оформлення програмного тексту. Головна вимога — це послідовність. Суміш елементів кількох стилів в одному тексті — це вже відсутність стилю. Обравши один раз певні правила, слідувати їм всюди в тексті даного програмного продукту (або, якщо нагальна потреба змінити стиль виникла вже в процесі роботи над програмою, треба не полінуватися й переоформити вже написану частину).

В.1. Імена змінних та функцій

Імена змінних та функцій повинні бути осмисленими, щоб при першому погляді на ім'я було зрозуміло, який сенс цієї змінної чи функції, яку роль вони відіграють у програмі чи яку задачу вирішують.

Однолітерні імена (`n`, `x`, `y` тощо) можна застосовувати лише для змінних і лише тоді, коли постановка задачі сама по собі математична, і імена змінних у програмі є прямими відповідниками позначень у математичних викладках. Крім того, імена `i`, `j` типові для лічильників у циклі **`for`**.

В усіх інших випадках є сенс давати змінним та функціям імена, що складаються зі слова або кількох слів та описують їх сенс.

Для імені змінної частіше за все беруть іменник, наприклад `length` (довжина), `count` (лічильник), `password` (пароль). Для імені функції краще підходить дієслово: `decode` (розкодувати), `draw` (намалювати), `transmit` (передати). Хоча в деяких випадках сенс функції добре відображає іменник: так, функція `time` повертає час у секундах, що пройшов від початку Ери UNIX (00:00:00 1 січня 1970 року) до поточного моменту. Краще, щоб ім'я змінної чи функції починалося з малої літери.

Іноді одного слова буває недостатньо, щоб виразити сенс змінної чи функції. Тоді потрібно зіставляти ім'я з кількох слів. Наприклад, для функції, яка має повертати визначник матриці, ім'я буде складатися з англійських слів *get matrix determinant*. Тут можливі три підходи:

1. всі слова писати малими літерами підряд: `getmatrixdeterminant`;
2. розділяти слова знаками підкреслювання: `get_matrix_determinant`;
3. писати всі слова разом, але починати кожне слово (крім першого) з великої літери: `getMatrixDeterminant`.

На даний момент найбільше прихильників завоював третій спосіб: він не збільшує довжину ідентифікаторів, як другий, та разом з тим не вимагає вчитуватися в кожну літеру довгого ідентифікатора. Кожним з трьох стилів написано безліч програм. Головне — обравши один з них, чітко дотримуватися саме його (принаймні, в межах одного проекту), бо змішування кількох стилів в одній програмі може лише заплутати програміста: йому буде важко згадати, в якому стилі названо ту чи іншу функцію.

Крім того, в іменах змінних і функцій прийменник to (до) часто замінюють цифрою 2, а прийменник for (для) цифрою 4 через подібність вимови (two, four). Наприклад: `string2int` (таке ім'я може мати функція, що перетворює текстовий рядок, що складається з цифр, на відповідне ціле число).

Імена структурних типів часто починають з літери «T», наприклад: `TRational`, `TFigure`, `TVector`. Імена макросів часто пишуть великими літерами: `BUFFER_SIZE`, `DENSITY`.

B.2. Оформлення виразів

Перед і після знаку бінарної операції повинні стояти пробіли, щоб відкремити знак операції від операндів. Інакше більш-менш складний вираз зливається в суцільну мішанину символів, і людині треба напружувати очі та витрачати додатковий час, щоб розібратися у виразі. Порівняйте, наприклад, два способи написання виразу:

```
p=(i*n+j)%m;
p = (i * n + j) % m;
```

Іноді буває зручно залишати у тексті виразу «зайві» дужки. Такі дужки не впливають на обчисленння виразу, бо і без них вираз обчислювався б у тій же послідовності завдяки пріоритетам операцій, але роблять вираз зрозумілішим для людини (тим більше, що людина може й не тримати в голові всю таблицю пріоритетів операцій і сумніватися в деяких виразах). Наприклад, такий вираз незручний для сприйняття:

```
alert = ! safe && r < 10 || x == y;
```

Унарна операція `!` (заперечення) має найвищий пріоритет, тому першим обчислюється підвираз `!safe`. Другим обчислюєтья підвираз `r<10`, третім — кон'юнкція двох цих значень. Нарешті, обчислюється вираз `x==y` та диз'юнкція його значення з попереднім. Але людина, яка читає текст програми, може засумніватися, відноситься заперечення до однієї лише змінної `safe` чи до всього виразу, а також який підвираз є другим членом кон'юнкції: `r<10` чи може `r<10 || x==y`. Тому набагато зрозумілішим виглядатиме текст, оформлений наступним чином:

```
alert = ((! safe) && (r < 10)) || (x == y);
```

При виклику функції, яка має кілька аргументів, аргументи потрібно розділяти пробілами. Наприклад, другий рядок читається зручніше за перший:

```
s = pow(x,p);
s = pow(x, p);
```

B.3. Відступи

Оператори, що складають тіло деякої функції, не всі рівноправні між собою, з точку зору логіки побудови та функціонування програми вони утворюють ієрархію. Одні оператори працюють під управлінням інших, наприклад, оператор може стояти в тілі циклу. Щоб логіку програми можна було зрозуміти, не вчитуючись в кожен оператор, розташування рядків програмного тексту повинно висвітлювати цю структуру.

Оператори, що підпорядковані деякому іншому оператору, потрібно набирати з відступом на певну кількість пробілів (часто на два). Якщо в операторі, що сам підпорядкований іншому оператору, є, в свою чергу, підпорядковані оператори, їх треба набирати з подвійним відступом, і так далі. Нижче наведено приклади двох способів оформити один і той самий програмний текст, другий спосіб, звичайно ж, зрозуміліший.

```
/* негарно */
int n, i, j;
printf( "Введіть␣розмір␣трикутника␣" );
scanf( "%d", n );
for( i = 1; i < n; i++ ) {
for( j = 1; j <= n-i+1; j++ )
printf( "*" );
printf( "\n" );
}

/* гарно */
int n, i, j;
printf( "Введіть␣розмір␣трикутника␣" );
scanf( "%d", n );
for( i = 1; i < n; i++ ) {
  for( j = 1; j <= n-i+1; j++ )
    printf( "*" );
  printf( "\n" );
}
```

В другому випадку видно, наприклад, що цикл по змінній j стоїть у тілі іншого циклу по змінній i, друк зірочки підпорядкований внутрішньому циклу, а друк символу переходу на новий рядок — зовнішньому.

Деякі знані спеціалісти радять відкривальну фігурну дужку розміщувати в окремому рядку, строго під першою літерою слова, до якого вона належить. Закривальну дужку вони теж радять вміщувати на окремому рядку, строго під відповідною відкривальною дужкою. Наприклад, два вкладених цикли при такому оформленні виглядали б так:

```
for( i = 1; i < n; i++ )
{
  початок тіла зовнішнього циклу
  for( j = 1; j <= n-i+1; j++ )
  {
    тіло внутрішнього циклу
  }
  решта тіла зовнішнього циклу
}
```

Такий спосіб оформлення програмних текстів набув популярності, деякі розповсюджені середовища програмування навіть автоматично форматують текст в процесі введення саме таким чином. Але автор даної книги вважає, що виділяти окремий рядок тексту під одну лише відкривальну дужку — значить збільшувати довжину тексту, після чого може стати важче охопити одним поглядом зовнішній цикл. Натомість зручніше розміщувати відкривальну фігурну дужку одразу після ключового слова, в тому ж рядку, а закривальну ставити в окремому рядку, строго під першою літерою відповідного ключового слова. Всі приклади в цьому посібнику оформлено саме в такому стилі.

Все сказане в цьому розділі про відступи операторів в тілах функцій можна застосувати і до полів структурних типів. Крім того, рекомендується вирівнювати типи, імена полів структури та коментарі до них, щоб вони утворювали колонки, як в таблиці:

```
typedef struct tagVector {
  int    allocated;   /* виділено пам'яті */
  int    used;        /* реально задіяні */
  double *components; /* динам. масив */
} SVector;
```

B.4. Оформлення тексту в цілому

Транслятору все одно, в якому порядку стоять у програмі директиви підключення заголовочних файлів, оголошення функцій, макросів тощо — головне, щоб кожен ідентифікатор був оголошений перед першим використанням. Але щоб не ускладнювати роботу з текстом, варто розташовувати його «смислові блоки» в чітко визначеному порядку. Смислові блоки треба відділяти один від одного кількома (2–4) порожніми рядками.

Текст кожного заголовочного файлу чи модуля програми повинен починатися з великого коментаря, який містить: назву програмного продукту, його анотацію (короткий опис призначення та основних функціональних можливостей), ім'я і контактні дані (адреса електронної пошти, номер ICQ) автора, рік розробки. Не слід шкодувати часу на оформлення такого вступного коментаря до кожного файлу: по мірі того, як у програміста з роками накопичуються сотні файлів, йому самому стає в них дедалі важче орієнтуватися. Між вступним коментарем та рештою тексту потрібно пропустити кілька порожніх рядків.

Далі треба розмістити директиви підключення заголовочних файлів. Спочатку йдуть стандартні заголовочні файли, потім, через порожній рядок, власні. Після директив підключення треба знов пропустити кілька порожніх рядків.

В більшості випадків можна оголосити спочатку всі макроси, потім всі типи даних, потім всі глобальні змінні (якщо вони є — нагадаємо ще раз, що використання глобальних змінних треба будь-що уникати), всі прототипи функцій і, нарешті, описати всі тіла функцій. Кожна з перерахованих частин відокремлюється порожніми рядками.

До оголошень макросів, типів даних, змінних, функцій потрібно обов'язково писати коментарі. Оголошення макросів та змінних зазвичай бувають не надто довгими, тоді коментар рекомендується вмістити в той же рядок. Прототипи функцій краще коментувати окремим рядком, перед самим прототипом.

Часто макроси та змінні за своїм сенсом та призначенням розпадаються на логічно пов'язані групи (наприклад: група макроозначень кодів клавіш, окремо група макроозначень кодів кольорів, група макроозначень розмірів масивів), тоді такі групи варто відділяти одна від одної порожнім рядком та перед кожною з них писати коментар. Крім того, варто в таких випадках робити імена макроконстант за єдиним зразком — вони мають починатися з префіксу, який дозволяє людині одразу побачити, до чого відносяться ці константи:

```
/* коди клавіш */
#define KEY_ENTER 13
#define KEY_ESC   27
#define KEY_SPACE 32

/* кольори елементів інтерфейсу */
#define COLOR_MENU  7  /* меню */
#define COLOR_TEXT  14 /* звичайний текст */
#define COLOR_PANEL 1  /* фон тексту */
```

Завдяки префіксам `KEY` та `COLOR` людині буде легко відрізнити імена тих макроконстант, що відносяться до кодів клавіш, від макроконстант для кольорів.

Частіше за все функції, що складають програму чи модуль, також можна природним чином об'єднати в групи за призначенням (функції для роботи з матрицями, функції для

оформлення інтерфейсу користувача, функції для роботи з «мишею» тощо). Оголошення споріднених функцій треба завжди групувати між собою. Імена функцій, які здійснюють споріднені, близькі між собою за сенсом та призначенням дії, також бажано починати з однакового префіксу (у прикладі далі імена функцій для роботи з матрицями починаються з префіксу `mtx`, імена функцій для обробки списків — з префіксу `lst`). Перед кожною такою групою потрібно писати загальний коментар.

```
/**** функції для обробки матриць ****/
/* додавання */
void mtx_add( double **a,
  double **b, double **p, int );
/* множення */
void mtx_mult( double **a,
  double **b, double **p, int );
/* обернена */
void mtx_inv( double **a, double **p, int );

/**** функції для обробки списків ****/
/* дізнатися довжину */
int lst_length( TList *p );
/* знайти елемент за номером */
TList  *lst_seek( TList *p, int i );
/* видалити елемент після даного */
void lst_del( TList *p );
```

Після всіх оголошень повинні йти означення функцій. Правила гарного стилю для їх оформлення описано в наступному розділі.

B.5. Стиль означень функцій

Тіла функцій треба відділяти одне від одного принаймні двома порожніми рядками. Перед тілом кожної функції потрібно писати детальний коментар, приклад якого показано нижче. В такому коментарі пишуть:

- призначення функції, короткий опис задачі, яку вона розв'язує. Наприклад: обчислення середнього арифметичного елементів масиву, обчислення оберненої матриці, сортування масиву за спаданням тощо;
- сенс її аргументів. Наприклад, якщо функція має аргумент `n` типу **`int`**, то в коментарі обов'язково треба написати, що це за число — кількість елементів якогось масиву, якась відстань у метрах, оцінка з якогось предмету чи щось інше;
- передумови, тобто умови щодо значень аргументів, при яких її виклик має сенс, іншими словами — умови, яким повинні задовольняти дані, що подаються на вхід функції. Наприклад, якщо функція має аргумент `n` типу **`int`**, сенсом якого є кількість елементів масиву, то, звичайно ж, повинна бути передумова $n > 0$.
- сенс значення, яке повертає функція. Наприклад, в коментарі до функції, яка шукає в масиві задане значення, може бути написано: «повертає номер k такого елементу масиву, що p[k]==x, а якщо такого елементу не існує, повертає число -1».
- побічні ефекти. Побічними ефектами називають будь-які результати роботи функції окрім повернення значень. Наприклад: «функція друкує на екран матрицю» або «дописує в кінець файлу один рядок тексту». Окремим різновидом побічних ефектів, який часто використовується на практиці, є присвоєння значень змінним, покажчики на які передані у функцію в якості аргументів. Наприклад, стандартна функція `strcpy` має два аргументи типу покажчиків на масиви символів, і результатом її роботи стає присвоєння значень з другого масиву елементам першого масиву. Зазначимо, що термін «побічний ефект» не треба розуміти як «другорядний, неважливий ефект» (в прикладі

з функцією `strcpy` це зовсім не другорядний, а саме основний результат, заради якого функція і використовується) — це означає «результат, який досягається не поверненням значення».

Приклад:

```
/*************************************
mtx_inv - обчислення оберненої матриці
*** Аргументи:
double **a - яку матрицю обертати
double **p - куди присвоїти результат
int n - розмірність матриці
*** Передумови:
* a та p мають вказувати на квадратні
матриці (n рядків по n елементів);
* матриця a повинна бути невиродженою
(визначник не 0), інакше оберненої
не існує.
*** Повертає:
1, якщо обчислити обернену вдалося;
0, якщо матриця вироджена
*** Ефекти:
якщо матриця a невироджена, то в
масив p присвоюються елементи її
оберненої матриці
*************************************/
int mtx_inv(double **a,
  double **p, int n) {
. . .
}
```

Додаток Г

Словник термінів

В словнику подано не всі, а лише найважливіші або найскладніші для розуміння терміни. Цифри в дужках наприкінці роз'яснення терміну означають розділ, в якому міститься детальніша інформація про дане поняття. Якщо в роз'ясненні терміну використовуються інші терміни, включені в цей словник, вони виділяються *курсивом.*

Алгоритм — сукупність елементарних операцій та правил, що визначають послідовність їх виконання. Алгоритм — це уточнений до деталей спосіб досягнення певного результата (скажімо, алгоритм Евкліда для пошуку найбільшого спільного дільника двох чисел, бульбашковий алгоритм сортування масиву тощо). *Програма* — текст, написаний *мовою програмування*, є описом алгоритму, зрозумілим для обчислювальної машини, на відміну від, наприклад, словесного опису того ж алгоритму, зрозумілого для людини. (1.1, 1.3, 1.4)

Аргумент — значення, яке передається до *функції* при її *виклику*, а також спеціальна *локальна змінна* функції, через яку це значення передається. (5.1)

Багатовимірний масив — *масив*, елементами якого є, в свою чергу, масиви (на противагу одновимірним масивам, у яких елементами є прості об'єкти даних, наприклад, дійсні чи цілі числа). Найрозповсюдженішим частковим випадком є матриця — двовимірний масив, тобто масив, кожен з n елементів якого є масивом з m елементів (здебільшого, чисел). В мові C багатовимірний масив прийнято моделювати як одновимірний масив, елементами якого є покажчики на масиви. (7.8)

Бінарна операція — операція, яка має два операнди. Наприклад, арифметичні операції додавання, віднімання, множення та ділення: у виразі $a+b$ операція додавання здійснюється над двома аргументами, a та b. У мові C бінарними операціями також є: залишок від ділення `%`, логічні операції кон'юнкція `&&` та диз'юнкція `||`, операції порівняння `==`, `!=`, `>` та деякі інші.

Виклик функції — різновид оператору мови C; процес виконання цього оператору. Нехай в програмі оголошено *функцію* `f`, яка має *аргументи* `x1,...,xn` типів (див. *тип даних*) `T1,...,Tn` відповідно. Тоді оператор виклику функції `f`, який може бути розташований в будь-якій функції `g`, має вигляд `f(E1,...,En)`, де `E1,...,En` — вирази типів `T1,...,Tn`. Іншими словами, оператор виклику складається з імені функції та списку виразів, значення яких мають бути передані в якості аргументів. Процес виклику відбувається таким чином. Спочатку обчислюються значення виразів `E1,...,En`, ці значення присвоюються змінним `x1,...,xn` функції `f`, далі локальні змінні функції `g` запам'ятовуються та приховуються, а локальні змінні функції `f` створюються. Тимчасово припиняється виконання функції `g` (обчислювальна машина запам'ятовує, на якому саме місці воно перерване) та починається виконання операторів функції `f`. Після цього виконується повернення у функцію `g`: локальні змінні функції `f` знищуються, відновлюються приховані локальні змінні функції `g`. Відновлюється призупинене виконання функції `g`, причому значення, яке стало результатом виконання функції `f` (якщо воно є), надходить у функцію `g` як значення виразу `f(E1,...,En)`. (5.4)

Глобальна змінна — в мові C *змінна*, оголошена не в якій-небудь *функції*. *Час життя* такої змінної — весь час виконання програми, *область видимості* — тіла всіх функцій, розташовані після оголошення змінної. Це означає, що глобальна змінна доступна кільком функціям: якщо одна функція присвоює такій змінній значення, то інша може використати це значення. Тому функції, які спільно використовують глобальну змінну, виходять сильно зв'язаними між собою: помилка чи зміна в одній з них відіб'ється

на другій. Через це програми, що містять глобальні змінні, важко проектувати, розуміти та відлагоджувати, глобальні змінні роблять логічну структуру програми надто складною. В сучасних підходах до програмування рекомендується не використовувати глобальні змінні. Для інтерфейсу між функціями натомість рекомендується використовувати передачу *аргументів.* (5.3)

Динамічне управління пам'яттю — управління пам'яттю, яке *програма* здійснює в процесі виконання. Програма по ходу виконання вирішує, скільки байтів пам'яті їй потрібно, та звертається до операційної системи чи системи підтримки виконання програми з запитом на виділення області пам'яті потрібного розміру. Виділивши у вільній пам'яті область, система повідомляє програмі адресу, з якої ця область починається (покажчик на початок області). Знаючи цю адресу, програма може розміщувати за нею свої дані. По закінченні роботи з цими даними програма, як правило, повинна звільнити область пам'яті — повідомити системі, що ця область пам'яті більше не потрібна, щоб система знов розглядала її як вільну пам'ять. В мові С динамічне управління пам'яттю частіше за все використовується для обробки *масивів*, розмір яких заздалегідь невідомий. (7.5)

Змінна — іменована комірка пам'яті. Має певний *тип.* В кожен момент часу містить певне значення цього типу. *Оператор присвоювання* надає змінній нове значення. Таким чином, в процесі роботи програми змінна може набувати то одного, то іншого значення. Поточне значення змінної може використовуватися при обчисленні виразів. В мові С кожна змінна обов'язково повинна бути оголошена. Оголошення повідомляє *компілятору* ім'я та тип змінною. Будь-які операції в програмі можливі лише з оголошеними змінними. Див. також: *глобальна змінна*, *локальна змінна*, *область видимості змінної*, *час життя змінної.* (1.2, 1.4, 2.1, 2.3)

Інтерпретатор — *транслятор*, який виконує *програму*, написану *мовою програмування високого рівня*, в процесі обробки її тексту, на відміну від *компілятора*, який, не виконуючи програми, будує її переклад *машинною мовою.* Перевагою інтерпретаторів є гнучкість — програміст може втрутитися у процес виконання програми та, можливо, внести зміну у програму, не перериваючи її виконання. Недоліком інтерпретаторів порівняно з компіляторами є невисока швидкість виконання програм. Інтерпретаторами є, наприклад, більшість трансляторів мови Бейсік. (1.1)

Компілятор — *транслятор*, який перетворює весь текст вихідної *програми* з *мови програмування високого рівня* в послідовність команд *машинної мови.* Компілятор, на відміну від *інтерпретатора*, не виконує вихідну програму, а лише перекладає її машинною мовою. Отримана в результаті цього програма в машинних кодах може потім безпосередньо (без застосування транслятора чи інших допоміжних засобів) виконуватися на обчислювальній машині. Витративши один раз час на компіляцію, можна отримати високоефективний код, який швидко виконується. Транслятори мови С є саме компіляторами. (1.1)

Локальна змінна — в мові С *змінна*, оголошена в якій-небудь *функції.* Її *областю видимості* є тіло цієї ж функції, *часом життя* — той проміжок часу, протягом якого ця функція виконується. Перше означає, що локальну змінну «бачать» лише оператори, розташовані в тілі цієї функції, друге — що локальна змінна створюється в момент *виклику* даної функції та знищується при *поверненні* з неї. Отже, робити будь-які дії з локальною змінною може лише та функція, в якій ця змінна оголошена, будь-які інші функції не мають до цієї змінної жодного доступу. Локальність змінних добре підтримує декомпозицію — дозволяє програмісту, який розробляє деяку функцію, не брати до уваги, з яких інших функцій буде викликатися його функція, тобто дозволяє зробити функції максимально незалежними одна від одної. (5.3)

Масив — складний об'єкт даних, послідовність, що складається з фіксованої кількості об'єктів однакового типу (елементів масиву). До кожного елементу масиву можна звернутися за його порядковим номером. Операція, яка за заданим номером вибирає з масиву

відповідний елемент, називається індексуванням, номер елементу прийнято називати індексом. Важливо запам'ятати, що нумерація елементів масивів в мові C починається з 0, тобто якщо масив складається з N елементів, то елементи мають номери з 0 по $N-1$. Оскільки індекс в тексті програми може бути заданий змінною (звичайно, цілого типу), з'являється можливість втілити в програмах такий надзвичайно важливий на практиці спосіб роботи з даними, як поелементна обробка масиву, тобто застосування однієї операції або алгоритма до кожного по черзі елементу масиву. В мові C поняття масиву тісно пов'язане з поняттям *покажчика*: оскільки елементи масиву розташовуються в пам'яті підряд, то для обробки будь-якого елементу масиву природно використовувати покажчик на його перший елемент, а індекс розглядати як зміщення від першого елементу. Виключно через покажчики обробляються в мові C такі масиви, пам'ять під які виділяється динамічно (див. *динамічне управління пам'яттю*). Див. також *багатовимірний масив.* (6)

Машинна мова — *мова програмування*, яка складається з команд, спосіб виконання яких (*семантика*) реалізований в обчислювальній машині на рівні електронної схеми. Інша назва — мова низького рівня. *Програма*, написана такою мовою, може бути безпосередньо виконана машиною. Іншими словами, машинна мова є зрозумілою для машини. Водночас, машинні мови майже непридатні для розуміння та написання програм людиною. Див. також *мова високого рівня.* (1.1)

Мова високого рівня — *мова програмування*, у якої *синтаксис* та *семантика*, логічна структура спроектовані таким чином, щоб мовою якнайзручніше було користуватися людині. Як правило, мови високого рівня підтримують деякі прийоми людського мислення; в ідеалі процес написання тексту *програми* мовою високого рівня повинен бути природним продовженням розмірковувань над задачею та продовженням звичайного викладу думок про задачу. Розроблені на даний час мови в більшій чи меншій мірі наближаються до цієї мети. Будучи орієнтованою на людину, мова високого рівня не забезпечує безпосередньої зрозумілості текстів програм для обчислювальної машини. Тому для виконання на машині написаний людиною текст програми обробляється за допомогою *транслятора*, який або сам виконує всі описані в тексті програми дії (*інтерпретатор*), або перекладає програму машинною мовою (*компілятор*), щоб потім перекладену програму можна було виконувати на машині. (1.1)

Мова програмування — штучна мова, призначена для такого опису *алгоритмів*, який придатний для подальшого виконання цих алгоритмів на обчислювальній машині. Виділяються *мови програмування високого рівня*, зручні та зрозумілі для людини, та *машинні мови*, пристосовані для безпосереднього записаних ними текстів обчислювальною машиною. (1.1)

Мова низького рівня — див. *Машинна мова*

Область видимості змінної — та частина програмного тексту, з якої допустимі звертання до даної *змінної.* (5.3)

Оператор циклу — такий *оператор*, який примушує обчислювальну машину повторювати певну дію декілька разів, в найзагальнішому випадку — невизначено багато разів, поки справджується деяка умова. Розрізняють оператори циклу з передумовою, з постумовою та з параметром. Оператор циклу з передумовою складається з умови b та оператору S, який називають тілом циклу. На кожній ітерації перевіряється умова b: якщо вона хибна, цикл закінчується, а якщо істинна, то виконується тіло S та процес повторюється знов. Оператор циклу з постумовою відрізняється тим, що перевірка умови здійснюється не до, а після кожного виконання тіла. Оператор циклу з параметром зазвичай використовують, щоб по черзі перебрати всі значення з деякого діапазону чисел та для кожного з них виконати оператор S. (3.4, 3.5, 3.6).

Оператор — "речення" *мови програмування високого рівня*, визначає деяку дію, яку має виконати обчислювальна машина, або керує послідовністю виконання інших операторів. З операторів, як з простих будівельних блоків та з'єднань, будується *програма.*

Покажчик — тип даних, у якому значеннями є адреси деяких об'єктів, наприклад *змінних*, а серед операцій є т.зв. розіменування — операція, яка дозволяє за відомим покажчиком знайти відповідну змінну. Також покажчиком називають значення такого типу. Прийнято говорити, що покажчик вказує на об'єкт чи змінну. В мові C покажчики відіграють надзвичайно важливу роль, багато прийомів програмування на C основані саме на покажчиках. Особливо тісно пов'язані з *динамічним виділенням пам'яті* та *масивами.* (7)

Рекурсія — в програмуванні один з типових способів організації обчислювального процесу, при якому *функція* викликає сама себе. Доведено, що будь-яку програму, в якій використовується рекурсія, можна перетворити на еквівалентну програму, що не містить рекурсії — рекурсію можна замінити циклом. Навпаки, будь-який цикл можна замінити рекурсією. Логічній структурі мови C, колу задач, які традиційно програмуються цією мовою більш притаманні циклічні структури програм. Рекурсія в мові C потребує значних витрат пам'яті та має нижчу швидкодію, ніж цикли. Але деякі задачі за самою своєю природою більше пристосовані до розв'язку за допомогою рекурсії. Рекурсія внутрішньо притаманна, зокрема, таким задачам, які містять деякий параметр n, та в яких для того, щоб отримати розв'язок при деякому значенні $n > 0$ потрібно спочатку розв'язати допоміжну задачу для значення параметру $n - 1$. Наприклад, для того, щоб отримати факторіал числа n, можна спочатку знайти факторіал числа $(n-1)$, нехай це буде p, тоді відповідь легко отримати множенням $n! = n \cdot p$. Див. *рекурсивна функція.* (5.6)

Рекурсивна функція — *функція*, яка викликає сама себе. Розрізняють пряму рекурсію (в тілі функції `f` міститься виклик функції `f`) та непряму (в тілі функції `f1` є виклик функції `f2`, у тілі функції `f2` є виклик функції `f3`,..., нарешті у тілі функції `fn` викликається функція `f1`). Принципово важливо, щоб рекурсивна функція викликала себе з більш «простими» значеннями аргументів, ніж була викликана сама. Наприклад, для того, щоб функція «факторіал» обчислила значення для аргументу, що дорівнює n, вона повинна викликати сама себе зі значенням аргументу $n - 1$ (щоб обчислити факторіал числа n, потрібно спочатку обчислити факторіал числа $n - 1$, нехай це буде p, тоді відповіддю буде добуток $n \cdot p$). Також важливо, щоб для деяких (змістовно «найпростіших») значень аргументів значення функції обчислювалося без рекурсивного виклику. Наприклад, функція «факторіал» за одну дію, не викликаючи сама себе, обчислює значення, якщо аргумент дорівнює 0 — значенням функції є 1. Без цих двох умов спроба викликати функцію призведе не до отримання результату, а до теоретично нескінченного процесу викликання функцією самої себе (на практиці — до краху програми через вичерпання усієї вільної пам'яті). Див. також *рекурсія.* (5.6)

Тип даних — множина значень разом з множиною можливих операцій над ними та зі способом зберігання цих значень у пам'яті. Одне з найважливіших понять у програмуванні: як би кардинально не відрізнялися між собою *мови програмування*, в них обов'язково є ті чи інші типи даних і правила роботи з ними. В мові C кожна *змінна* обов'язково має один і тільки один тип. Тип кожної змінної вказується при її оголошенні. Завдяки цьому *транслятор* заздалегідь знає, які значення допустимо присвоювати цій змінній, та які операції над нею дозволяється робити. Крім того, тип змінної повідомляє транслятору, скільки потрібно виділити байтів пам'яті для зберігання змінної. У мові C для кожної *функції* повинен бути відомий тип значення, яке вона повертає, та типи всіх її *аргументів.* Це дозволяє транслятору перевірити правильність виклику функції (наприклад, дозволяє запобігти такій помилці, як виклик функції з аргументами неправильних типів). В мові C є набір стандартних типів (цілий та довгий цілий, дійсний, символьний, *покажчик*) та, що дуже важливо, засоби, які дозволяють користувачеві конструювати свої, як завгодно складні, типи даних. (1.2, 9)

Транслятор — спеціальна програма, яка обробляє текст *програми*, записаний *мовою програмування високого рівня*, роблячи його придатним для виконання обчислювальною

машиною. В процесі роботи транслятор також виявляє в тексті, що обробляється, синтаксичні та деякі семантичні помилки. Серед трансляторів виділяються два різновиди: *інтерпретатори* та *компілятори.* (1.1)

Умовний оператор — (інша назва — розгалуження) *оператор*, що примушує обчислювальну машину виконати один з двох заданих операторів в залежності від того, чи справджується певна умова. Умовний оператор складається з умови b та операторів S_1, S_2 і виконується таким чином: спочатку перевіряється умова b, якщо вона істинна, то далі виконується оператор S_1, інакше виконуєтсья оператор S_2. В мові С в ролі умови b використовується будь-який *вираз* числового типу; вважається, що умова справджується, коли обчислення виразу дає рузультат, відмінний від 0. Умовний оператор є програмною моделлю прийняття рішення в залежності від обстановки. (3.3)

Файл — іменована сукупність даних, що зберігаються на деякому пристрої (магнітному диску, компакт-диску, магнітній стрічці тощо). Частіше за все призначений для довготермінового зберігання даних. Наприклад, завдяки використанню файлів можна записати на пристрій проміжні результати роботи користувача з програмою, потім запустити програму ще раз, прочитати ці ж результати та продовжити їх обробку. Крім того, файли дозволяють одну й ту саму сукупність даних обробляти декількома програмами по черзі. Обробка файлів частіше за все відбувається послідовно. У файлі, що обробляється, є уявна голівка читання-запису, а сам файл поводить себе як довга стрічка з записаними підряд даними. При кожній операції голівка або читає дані, що знаходяться прямо перед нею, або записує дані в ту позицію на стрічці, біля якої розташована, просуваючись при цьому вперед. Операції введення та виведення над файлом з суто текстовою інформацією мають важливі особливості порівняно з загальним випадком, коли у файлі містяться довільні дані, закодовані двійковими числами. Тому у мові С розрізняються засоби обробки текстових та двійкових файлів. (10, 11)

Функція — об'єднані в єдине ціле оголошення змінних та оператори, що разом розв'язують деяку задачу; складова частина програми, що має власне ім'я, може багаторазово викликатися з різних місць програми (при цьому їй зазвичай передаються деякі дані для обробки — *аргументи*), результатом роботи зазвичай стає обчислення значення певного типу. Поняття функції є надзвичайно важливим для мови С. Будь-яка програма у мові С являє собою набір функцій, одна з яких (зі стандартним ім'ям `main`) є головною — виконання програми починається з неї, ісаме вона викликає всі інші функції. Одна з основних переваг такого стилю програмування полягає в тому, щоб один раз описати у вигляді функції типовий фрагмент програми та потім багато разів звертатися до нього (замість того, щоб кожен раз вписувати в програму один і той самий фрагмент тексту) — це робить текст програми компактнішим та дозволяє легко вносити в нього зміни. Оскільки функцію, один раз написану, можна використовувати необмежену кількість разів, оформлення типових фрагментів програм у вигляді функцій значно підвищує продуктивність програмістської праці. Функції це не лише один з технічних засобів програмування, а основа стилю програмування у мові С: в процесі розгортання задуму від постановки задачі до готової програми програміст мусить розмірковувати в термінах функцій. Справді світоглядне значення поняття функції полягає в тому, що функції дозволяють створювати програму за методом «розділяй та володарюй» — велику та складну задачу можна розділити на підзадачі, кожну з них окремо та незалежно від інших підзадач розв'язати за допомогою окремої функції, а взаємодію між цими функціями (тобто збирання розв'язку цілої задачі з розв'язків підзадач) покласти на ще одну функцію. Якщо деякі з підзадач виявляться, в свою чергу, достатньо складними, то для їх розв'язання треба застосувати такий же підхід. Див. також: *виклик функції*, *локальна змінна.* (5.1)

Час життя змінної — проміжок часу, протягом якого дана *змінна* існує в пам'яті. Див. також: *глобальна змінна*, *локальна змінна.* (5.3)

Додаток Д

Найважливіші слова мови C

В словнику подано найважливіші слова мови C, які найбільше використовуються в практичному програмуванні та бездоганне знання яких абсолютно необхідне для програміста: оператори, імена стандартних функцій, директиви препроцесора, імена стандартних типів тощо. Цифри в дужках наприкінці роз'яснення терміну означають розділ, в якому міститься детальніша інформація.

break — оператор, який може стояти лише в тілі циклу або всередині оператору `switch`. В першому випадку розриває цикл (навіть якщо умова продовження циклу все ще істинна), в другому випадку перериває виконання оператора `switch`. (3.5, 3.7)

case — ключове слово, яке може використовуватися лише всередині оператору `switch` та позначає в ньому початок гілки. Після слова `case` пишеться константа, з якою порівнюється значення виразу, вказаного у заголовку оператору `switch`. Далі, через двокрапку, пишуться оператори, які будуть виконуватися, якщо значення виразу співпадає з даною константою. (3.7)

char — стандартний тип даних, використовується для двох цілей: як цілочисельний тип з короткою розрядною сіткою (один байт), придатний для обробки невеликих чисел, та як спеціалізований тип для роботи з числовими кодами символів (таких як літери, знаки пунктуації тощо). Текстові рядки в мові C моделюються як масиви значень типу `char`: кожен елемент масиву зберігає числовий код одного символу рядка. (8.1)

continue — оператор, який може міститися лише в тілі циклу. Виконання цього оператору призводить до негайного завершення поточної ітерації циклу та переходу до перевірки умови продовження циклу. (3.5)

default — ключове слово, яке може міститися лише в операторі `switch` та позначає гілку, яка виконується тоді, коли жодна з констант, вказаних у попередніх гілках `case` не співпадає зі значенням виразу у заголовку оператору `switch`. Іншими словами, це гілка, яку оператор `switch` виконує тоді, коли він не обирає якусь іншу гілку. (3.7)

define — директива препроцесору (перед кожною директивою повинен стояти символ `#`), яка слугує для означення макросів. Після даної директиви пишеться ім'я макросу і далі означення (будь-яка послідовність символів). Транслятор, читаючи подальший текст, всюди, де в тексті зустрічається дане ім'я макросу, буде замість нього підставляти відповідне означення. (6.1)

do — ключове слово, з якого починається оператор циклу з постумовою. Після даного слова слідує тіло циклу — оператор (чи послідовність операторів, взята у фігурні дужки) і далі слово `while` та в круглих дужках умова продовження циклу. Виконавши спочатку тіло, оператор циклу з постумовою перевіряє умову та, якщо вона істинна (числовий вираз дає результат, відмінний від 0), повторює цю дію знов. (3.4)

double — стандартний тип даних — дійсні числа з подвійною точністю. (2.3)

else — ключове слово, яке може використовуватися лише в умовному операторі (тобто разом з ключовим словом `if`). Після даного слова пишеться оператор (чи послідовність операторів у фігурних дужках), який виконується у випадку, коли умова даного умовного оператору хибна, тобто коли числовий вираз, записаний у дужках після слова `if` дає значення 0. (3.3)

fclose — стандартна функція (з файлу `stdio.h`), яка закриває файл. Має один аргумент типу покажчика на структуру `FILE` (яка відіграє роль каналу доступу до певного файлу,

в даний момент відкритого для читання або запису). В результаті виклика функції файл закривається, тобто припиняється обмін даними між файлом та програмою. (10.2)

feof — стандартна функція (з файлу `stdio.h`), яка дізнається, чи досягнуто кінець файлу в процесі його читання. Має один аргумент типу покажчика на структуру `FILE` (яка відіграє роль каналу доступу до певного файлу, в даний момент відкритого для читання або запису). Якщо поточна позиція читання-запису знаходиться наприкінці файлу, функція повертає число 1, інакше 0. (10.3)

fgets — стандартна функція (з файлу `stdio.h`), яка з відкритого для читання текстового файлу читає рядок (тобто послідовність символів до першого символу переходу на новий рядок) з обмеженням по довжині (тобто якщо навіть символ переходу на новий рядок не трапився, але кількість прочитаних символів перевищує задану межу, читання припиняється). Має три аргументи: покажчик на символьний масив, в який треба помістити прочитані з файлу символи, ціле число — максимально допустиму довжину рядка та покажчик на структуру `FILE` (яка відіграє роль каналу доступу до певного файлу, в даний момент відкритого для читання). (10.3)

FILE — структурний тип, оголошений у файлі `stdio.h`, призначений для роботи з файлами. Кожен об'єкт цього структурного типу містить деякі службові дані, за допомогою яких здійснюєтья звертання до того чи іншого файлу, розташованого на диску чи іншому пристрої. Іншими словами, програма взаємодіє з файлом не прямо, а через посередництво об'єкта типу `FILE`. Всі функції для файлового введення-виведення (читання та запис даних, переміщення по файлу тощо) мають аргумент типу покажчика на такий об'єкт. (10.1)

fopen — стандартна функція (з файлу `stdio.h`), яка відкриває файл. Має два аргументи: ім'я файлу, який потрібно відкрити (текстовий рядок) та режим відкриття (відкривати файл для читання, дописування в кінець чи для переписування заново тощо), також заданий текстовим рядком. В процесі відкриття файлу створює для нього відповідний об'єкт типу `FILE` (який надалі відіграє роль каналу для доступу до файлу) та повертає покажчик на цей об'єкт. (10.2)

for — ключове слово, означає оператор циклу з лічильником (з параметром). Після слова **`for`** в круглих дужках слідують три вирази, розділені між собою крапкою з комою і далі тіло циклу — оператор або послідовність операторів у фігурних дужках. Перший вираз виконується один раз на початку циклу, другий обчислюється на кожній ітерації перед виконанням тіла, третій вираз виконується наприкінці кожної ітерації, тобто після виконання тіла циклу. Якщо напочатку чергової ітерації другий вираз дає значення 0, цикл припиняється. В абсолютній більшості випадків перший вираз є присвоюванням деякій змінній `x` початкового значення (найчастіше 0), другий вираз має вигляд `x<N` (де `N` — деяка константа), третій вираз це `x++`. При виконанні такого циклу змінна `x` пробігає по черзі значення $0, 1, \ldots, N-1$. (3.6)

fprintf — стандартна функція (з файлу `stdio.h`), призначена для виведення даних, перетворених на текст, до файлу. Першим аргументом є покажчик на об'єкт структурного типу `FILE`, через посередництво якого здійснюється доступ до файлу, решта аргументів — форматний рядок та довільна кількість значень різних типів, як у функції `printf`. (10.3)

fputs — стандартна функція (з файлу `stdio.h`), призначена для виведення текстового рядка у файл. Перший аргумент — покажчик на текстовий рядок, який потрібно вивести, другий — покажчик на об'єкт структурного типу `FILE`, за допомогою якого програма отримує доступ до файлу. (10.3)

fread — стандартна функція (з файлу `stdio.h`), призначена для введення двійкових даних з файлу. При роботі з двійковими файлами обмін здійснюється цілими блоками однакового розміру (як правило, це об'єкти деякого структурного типу). Функція має 4 аргументи: покажчик на перший елемент масиву блоків, в який потрібно розмістити

блоки даних, прочитані з файлу; другий аргумент — розмір одного блока; третій — кількість блоків, які потрібно вивести; четвертий — покажчик на об'єкт структурного типу `FILE`, за допомогою якого здійснюється доступ до файлу. Повертає кількість успішно прочитаних з файлу блоків. (11.1)

free — стандартна функція (з файлу `alloc.h`), яка звільняє динамічно виділену область пам'яті. Має один аргумент — покажчик на область, яку треба звільнити. Див також: `malloc`. (7.5)

fscanf — стандартна функція (з файлу `stdio.h`), призначена для введення з файлу послідовності символів та перетворення її на дані певних типів. Перший аргумент — покажчик на структурний об'єкт типу `FILE`, за допомогою якого здійснюється звертання до файлу. Другим аргументом є форматний рядок, як у функції `scanf`, решта аргументів — покажчики на змінні, в які потрібно розмістити дані, отримані в результаті розбору символів, прочитаних з файлу. (10.3)

fseek — стандартна функція (з файлу `stdio.h`), яка дозволяє програмі довільно переміщувати голівку читання-запису по файлу. Перемістити голівку можливо: на n-й байт від початку файлу, на n-й байт від кінця файлу, на n-й байт від поточної позиції (в останньому випадку n може бути як додатнім так і від'ємним, що відповідає руху відповідно вперед та назад). Функція має ти аргументи: покажчик на об'єкт структурного типу `FILE`, через який програма має доступ до файлу; зміщення, тобто число байт (додатнє чи від'ємне), на яке треба перемістити голівку читання-запису; ціле число, яке позначає режим переміщення, один з трьох вказаних вище. (11.4)

ftell — стандартна функція (з файлу `stdio.h`), яка дозволяє програмі дізнатися, на якому від початку файлу байті знаходиться голівка читання-запису. Має один аргумент, покажчик на об'єкт структурного типу `FILE`, через який програма має доступ до файлу. Повертає номер поточної позиції у файлі. (11.4)

fwrite — стандартна функція (з файлу `stdio.h`), призначена для виведення двійкових даних у файл. При роботі з двійковими файлами обмін здійснюється цілими блоками однакового розміру (як правило, це об'єкти деякого структурного типу). Функція має 4 аргументи: покажчик на перший елемент масиву блоків, розмір одного блока, кількість блоків, які потрібно вивести, та покажчик на об'єкт структурного типу `FILE`, за допомогою якого здійснюється доступ до файлу. Повертає кількість успішно записаних у файл блоків. (11.1)

if — ключове слово, яке позначає умовний оператор. Після слова **`if`** в круглих дужках повинна стояти умова — деякий вираз і оператор чи послідовність операторів, взята у фігурні дужки. Далі може бути слово **`else`** та інший оператор чи у фігурних дужках послідовність операторів. Якщо умова істинна (обчислення виразу дає ненульове значення), виконується перший оператор (група операторів). В протилежному випадку виконується гілка після слова **`else`** (якщо вона є). (3.3)

int — стандартний тип даних, цілі числа з обмеженою довжиною розрядної сітки, на більшості процесорів 16 біт. При довжині 16 біт дозволяє обробляти числа з діапазону $-2^{15} \dots 2^{15} - 1$, тобто $-32768 \dots 32767$. Стандартні операції над значеннями цього типу — додавання, віднімання, множення, ділення (ділення цілого числа на ціле число дає результат також цілого типу), залишок від ділення, побітові кон'юнкція та диз'юнкція, побітові зсуви праворуч та ліворуч, деякі інші. Крім того, цілий тип у мові С використовується для моделювання логічного типу: логічному значенню «хиба» відповідає число 0, а значенню «істина» — будь-яке відмінне від 0 число. Для роботи з цілими числами в ролі логічних значень існують операції логічної кон'юнкції, диз'юнкції та заперечення. (2.3, 3.1, 3.2)

main — головна функція програми. В кожній програмі є рівно одна функція зі стандартним ім'ям `main`. Виконання програми починається саме з запуску цієї функції. Виклик будь-яких інших функцій, що входять до складу програми, здійснюється або прямо з функції

`main`, або з деякої іншої функції, яку прямо чи непрямо викликає функція `main`. Повертає ціле число — код завершення програми. Прийнято повертати число 0, якщо програма завершилася повністю успішно, інші числа — у випадку помилки. Цей код завершення може використовувати операційна система. (2.1, 5.1)

malloc — стандартна функція (оголошена в заголовочному файлі `alloc.h`), призначена для виділення області динамічної пам'яті певного розміру. Має один аргумент — розмір в байтах області пам'яті, яку потрібно виділити. Часто цю функцію використовують для динамічного створення масивів, тоді значення аргументу обчислюють як добуток розміру одного елементу масиву на кількість елементів. Повертає покажчик на створену в пам'яті область, якщо вдалося її створити. Якщо ж виділити область такого розміру неможливо (через брак вільної пам'яті), функція повертає порожній покажчик `NULL`. (7.5)

NULL — порожній покажчик, спеціальне значення в типі покажчика, яке означає, що даний покажчик не вказує на жодний певний об'єкт. (7.2)

printf — стандартна функція для форматованого виведення даних на екран (заголовочний файл `stdio.h`). Перетворює дані різних типів на послідовність символів за заданим шаблоном та виводить отримані символи на екран. Має змінювану кількість аргументів. Першим аргументом завжди повинен бути форматний рядок — рядок тексту, в якому може міститися довільна кількість специфікаторів формату. Кількість решти аргументів повинна бути такою самою, як і кількість специфікаторів (крім специфічних винятків), і типи цих аргументів повинні бути узгоджені зі специфікаторами. Кожен специфікатор пов'язаний з одним аргументом та визначає, як перетворювати зачення цього аргумента на символи. (2.6, 2.7)

return — ключове слово, яким позначається оператор повернення значення з функції. Після слова **`return`** повинен стояти деякий вираз того ж типу, який оголошено як тип значення функції. Коли виконання тіла функції доходить до цього оператору, обчислюється значення виразу, виконання функції завершується, знищуються всі її локальні змінні, а обчислене значення виразу стає результатом функції, тобто тим значенням, яке вона повертає туди, звідки була викликана. (5.1)

scanf — стандартна функція для введення даних різних типів з клавіатури (оголошена в заголовочному файлі `stdio.h`). Кількість аргументів змінювана. Першим аргументом завжди є форматний рядок, який може містити текст та будь-яку кількість специфікаторів формату. Решти аргументів повинно бути стільки ж, скільки й специфікаторів (крім деяких специфічних випадків). Кожен з цих аргументів повинен бути покажчиком на змінну, в яку потрібно розмістити прочитане з клавіатури значення. Функція читає символ за символом послідовність, введену користувачем з клавіатури. В процесі читання співставляє ці символи з символами та специфікаторами форматного рядка. Якщо чергові символи співставляються з черговим специфікатором, функція перетворює ці символи на відповідне значення та поміщає його в змінну, покажчик на яку міститься в черговому аргументі. Повертає число успішно співставлених специфікаторів. (2.8)

sizeof — операція, яка дозволяє дізнатися, скільки байтів займає об'єкт деякого типу. А саме, якщо `тип` — ім'я деякого типу даних, то значенням виразу **`sizeof`**`(тип)` є розмір об'єкту цього типу в байтах. Треба звернути особливу увагу, що **`sizeof`** це не функція, а операція (студенти часто припускаються помилки в цьому питанні через те, що круглі дужки при цій операції нагадують виклик функції). Принципова різниця в тому, що функція викликається під час виконання програми, а операція **`sizeof`** обробляється компілятором — дещо спрощено, компілятор замість виразу **`sizeof`** підставляє потрібне число. (7.5)

size_t — спеціальний цілочисельний тип даних, призначений спеціально для таких чисел, сенсом яких є розмір якогось файлу, розмір об'єкту в пам'яті тощо. Фактично, це не окремий тип, а інше ім'я для типу **`unsigned long`**. (11.1)

stdin — стандартний потік введення. Мова С дозволяє працювати з деякими пристроями так, ніби вони є файлами. Це означає, що через один і той самий набір функцій (`fprintf`, `fscanf` тощо) можна обмінюватися даними і з файлами на диску, і з такими пристроями, як екран, клавіатура, принтер та іншими. Змінна `stdin`, оголошена в файлі `stdio.h`, і є таким «файлом», що насправді пов'язаний з пристроєм введення (клавіатурою). (10.7)

stdio.h — заголовочний файл, в якому зібрано оголошення засобів введення-виведення: функцій `printf`, `scanf`, `fgets`, `fputs`, `fopen`, `fclose`, структурного типу `FILE`, потоків `stdin`, `stdout` та інших.

stdout — стандартний потік виведення — уявний файл, пов'язаний з пристроєм виведення (дисплеєм). Детальніше пояснення див. `stdin`. (10.7)

strcat — стандартна функція (оголошена в заголовочному файлі `string.h`), призначена для склеювання двох текстових рядків. Має два аргументи типу покажчиків на рядки. Перший аргумент — призначення (куди приєднувати), другий — джерело (що приєднувати). Дописує весь вміст другого рядка до кінця першого. Програміст сам відповідає за те, щоб в першому рядку було зарезервовано достатньо місця для символів, що будуть до нього дописано. (8.5)

strcmp — стандартна функція (оголошена в заголовочному файлі `string.h`), призначена для порівняння рядків. Має два аргументи — покажчики на рядки, які порівнюються. Повертає число 0, якщо рядки однакові, деяке від'ємне число, якщо перший рядок в словниковому порядку передує другому, та додатне число, якщо другий рядок передує першому. (8.5)

strcpy — стандартна функція (оголошена в заголовочному файлі `string.h`), призначена для копіювання рядків. Має два аргументи типу покажчиків на символьні рядки: перший — призначення (куди копіювати), другий — джерело (звідки копіювати). Копіює всі символи другого рядка у перший. Програміст сам відповідає за те, щоб у першому рядку було зарезервовано достатньо місця для символів, що копіюються. (8.5)

string.h — заголовочний файл, що містить оголошення стандартних функцій для обробки текстових рядків: `strlen`, `strcat`, `strcpy`, `strcmp` та багатьох інших. (8.5)

strlen — стандартна функція (оголошена в заголовочному файлі `string.h`), призначена для підрахунку довжини рядка. Має один аргумент — покажчик на рядок, довжину якого потрібно дізнатися. Повертає кількість символів в цьому рядку (без завершального нуль-символу). (8.5)

struct — ключове слово, яким позначається суктурний тип даних. Оголошення структурного типу має такий вигляд: слово **`struct`**, ім'я типу, далі у фігурних дужках список полів структури. Кожне поле оголошується так само, як і змінна: вказується тип та ім'я. Полями структури також можуть бути масиви та покажчики. В мові С (але не в С++) слово **`struct`** повинно стояти перед іменем структурного типу при кожному його використанні, скажімо, при оголошенні змінних структурного типу. (9)

switch — ключове слово, яким позначається оператор вибору. На відміну від умовного оператору, який дозволяє виконати в залаженості від істинності чи хибності умови одну з двох гілок, оператор вибору дозволяє вибрати гілку з багатьох в залежності від значення цілочисельного виразу. Після слова **`switch`** в круглих дужках пишеться вираз. Далі в фігурних дужках розміщується будь-яка кількість гілок, кожна з яких починається з ключового слова **`case`** та цілочисельної константи та містить оператори. Оператор вибору виконується таким чином: обчислюється значення виразу, потім отримане значення порівнюється з кожною по черзі константою, що стоїть при слові **`case`**. Якщо значення збігається, виконуються оператори, починаючи від даної гілки і до слова **`break`**. Оператор вибору може також містити особливу гілку, позначену словом **`default`** — вона виконується тоді, коли значення виразу не збігається з константами в жодній іншій гілці. (3.7)

typedef — ключове слово, за допомогою якого будь-якому типу даних можна надати власне ім'я. В простих випадках, що розглядалися в цьому посібнику, після слова `typedef` пишеться означення деякого типу даних (можливо, доволі довге, як наприклад означення структурного типу) і деякий ідентифікатор — нове ім'я типу. Всюди далі в тексті програми можна застосовувати це ім'я типу — компілятор автоматично буде під ним розуміти відповідне означення типу. (9.2)

void — порожній тип даних — спеціальний спосіб позначення, що по формі виглядає як тип даних, а означає відсутність даних будь-якого типу. Використовується як тип аргументу або значення функції: щоб позначити, що функція не має жодних аргументів, пишуть, ніби вона має аргумент типу `void`. Щоб позначити, що функція не повертає значення якого-небудь типу, пишуть, що функція ніби повертає значення типу `void`. (5.2)

while — ключове слво, що (1) позначає оператор циклу з передумовою або (2) стоїть наприкінці оператору циклу з постумовою. В першому випадку після слова `while` в круглих дужках стоїть умова продовження циклу та далі тіло циклу (оператор чи взята у фігурні дужки послідовність операторів). В другому випадку оператор циклу складається з ключового слова `do`, тіла та слова `while`, після якого в круглих дужках стоїть умова продовження. Умовою є вираз, якщо його обчислення дає значення, відмінне від 0, то цикл продовжується далі, в іншому випадку цикл завершується. Цикл з передумовою на кожній ітерації спочатку перевіряє умову продовження, потім виконує тіло, а цикл з постумовою спочатку виконує тіло, а потім перевіряє умову. (3.4)

Бібліоґрафія

[1] Абрамов С.А. и др. Задачи по программированию. — М.: Наука, 1988. — 280 с.

[2] Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979. — 536 с.

[3] Вайнер Р., Пинсон Л. C++ изнутри. Под ред. Хижняк И.В. — К.: ДиаСофт, 1993. —304 с.

[4] Вирт Н. Алгоритмы + структуры данных = программы. — М.: Мир, 1985. — 406 с.

[5] Глушаков С.В., Коваль А.В., Смирнов С.В. Язык программирования C++: Учеб. курс. — Харків: Фолио, 2001. — 500 с.

[6] Дейтел Х.М., Дейтел П.Дж. Как программировать на C. — М.: Бином, 2000. — 1008 с.

[7] Керниган Б., Ритчи Д. Язык программирования Си. — СПб.: Невский диалект, 2001. — 352 с.

[8] Кнут Д. Искусство программирования. — Т.3: Сортировка и поиск. — М.: Вильямс, 2004. — 703 с.

[9] Павловская Т.А. C/C++. Программирование на языке высокого уровня: Учебник. — СПб.: Питер, 2002. — 464 с.

[10] Сабуров С.В. Языки программирования C и C++. — М.: Познавательная книга плюс, 2001. — 656 с.

[11] Шилдт Г. Справочник программиста по C/C++. — К.: Вильямс, 2001. — 448 с.

ВІННИК Вадим Юрійович

Алгоритмічні мови та основи програмування: мова C

Навчальний посібник

Оформлено в системі LaTeX, макрос NCC

Редактор *Л.В. Гончарук*
Комп'ютерний набір та верстка *В.Ю. Вінник*
Макетування *В.В. Кондратенко*
Обкладинка *Д.В. Скачков*

Свідоцтво про державну реєстрацію КВ № 8079 від 30.10.2003. Підписано до друку 03.12.2007.
Формат $60 \times 84\frac{1}{16}$.
Гарнітура Computer Modern.
Друк офсетний. Ум. друк. арк. 20
Тираж 300 екз. Зам. 50

Редакційно-видавничий відділ
Житомирського державного технологічного університету
10005, м. Житомир, вул. Черняховського, 103