ФАКУЛЬТЕТ АВТОМАТИКИ, ТЕЛЕМЕХАНІКИ ТА ЗВ'ЯЗКУ

Кафедра «Спеціалізовані комп'ютерні системи»

В.С. Коновалов

ВИКОРИСТАННЯ МОВИ C^{++} ДЛЯ РОЗРОБЛЕННЯ СИСТЕМНИХ ПРОГРАМ

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни

"СИСТЕМНЕ ПРОГРАМУВАННЯ"

Харків 2011

Коновалов В.С. Використання мови С++ для розроблення системних програм: Конспект лекцій. – Харків: УкрДАЗТ, 2011. – 122 с.

Конспект лекцій призначений для вивчення принципів побудови програмних засобів з використанням мови високого рівня С++. Конспект лекцій підготовлений відповідно до програми дисципліни «Системне програмування».

Рекомендується для студентів спеціальності «Спеціалізовані комп'ютерні системи» всіх форм навчання.

Конспект лекцій розглянуто та рекомендовано до друку на засіданні кафедри «Спеціалізовані комп'ютерні системи» 27 листопада 2009 р., протокол № 3.

Рецензент

доц. А.В. Мамонов

В.С. Коновалов

ВИКОРИСТАННЯ МОВИ С++ ДЛЯ РОЗРОБЛЕННЯ СИСТЕМНИХ ПРОГРАМ

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни

"СИСТЕМНЕ ПРОГРАМУВАННЯ"

Відповідальний за випуск Коновалов В.С.

Редактор Ібрагімова Н.В.

Підписано до друку 25.01.10 р.

Формат паперу 60x84 1/16 . Папір писальний.

Умовн.-друк.арк. 3,75. Тираж 50. Замовлення №

Видавець та виготовлювач Українська державна академія залізничного транспорту
61050, Харків - 50, майдан Фейєрбаха, 7
Свідоцтво суб'єкта видавничої справи ДК № 2874 від 12.06.2007 р.

ЗМІСТ

1 ПРОЦЕДУРНЕ ПРОГРАМУВАННЯ

Історія програмування – це спроба подолати складність навколишнього світу. Перед програмістами встають усе більш складні задачі, способи їхнього розв’язання стають усе більш громіздкими, інформація, яку треба обробити, росте як сніжний ком. Ще нещодавно звичайними одиницями вимірювання інформації були кілобайти й мегабайти, а зараз уже говорять тільки про гігабайти й терабайти. Як тільки програмісти пропонують більш-менш задовільне розв’язання запропонованих задач, відразу виникають нові, ще більш складні задачі. Програмісти вигадують нові методи, створюють нові мови. За півстоліття з'явилося кілька сотень мов, запропоновано безліч методів і методик. Деякі методи й стилі програмування стають загальноприйнятими й утворюють на якийсь час так звану **парадигму** програмування.

1.1 ПАРАДИГМИ ПРОГРАМУВАННЯ

Перші, навіть найпростіші програми, написані в машинних кодах, становили сотні рядків зовсім незрозумілого тексту. Для спрощення й прискорення програмування створили мови високого рівня: FORTRAN, Algol та інші, поклавши рутинні операції з створення машинного коду на компілятор. Ті самі програми, переписані мовами високого рівня, стали набагато зрозумілішими й коротшими, але життя зажадало вирішення більш складних завдань, і програми, написані мовами високого рівня, знову збільшилися в розмірах, стали неозорими.

Виникла ідея: оформити програму у вигляді кількох по можливості простих процедур або функцій, кожна з яких вирішує своє визначене завдання. Написати, відкомпілювати й налагодити невелику процедуру можна легко й швидко. Потім лишається тільки зібрати всі процедури в потрібному порядку в одну програму. Крім того, один раз написані процедури можна потім використовувати в інших програмах як будівельні цеглинки. Процедурне програмування швидко стало парадигмою. В усі мови високого рівня включили засоби написання процедур і

функцій. З'явилася множина бібліотек процедур і функцій на всі випадки життя.

Постало питання про те, як виявити структуру програми, розбити програму на процедури, яку частину програми виділити в окрему процедуру, як зробити алгоритм розв’язання задачі простим і наочним, як зручніше пов'язати процедури між собою. Досвідчені програмісти запропонували свої рекомендації, названі структурним програмуванням. Структурне програмування виявилося зручним і стало парадигмою. З'явилися мови програмування, наприклад, `Pascal`, на яких зручно писати структурні програми. Більш того, на них дуже важко написати неструктурні програми. Мова С теж написана з урахуванням ідей структурного програмування. Оператори мови С дозволяють виявити структуру програми, чітко описати застосовувані в ній алгоритми.

Складність посталих перед програмістами задач проявилася і тут: програми стали вміщувати сотні процедур, знову стали неозорими. "Цеглинки" виявилися занадто малими. Потрібен був новий стиль програмування.

У цей же час виявилося, що вдала або невдала структура вихідних даних може сильно полегшити або ускладнити їхню обробку. Одні вихідні дані зручніше об'єднати в масив, для інших більше підходить структура дерева або стека. Ніклаус Вірт навіть назвав свою книгу: "Алгоритми + структури даних = програми".

Виникла ідея об'єднати вихідні дані й всі процедури їхньої обробки в один модуль. Ця ідея модульного програмування швидко заполонила розуми й на якийсь час стала парадигмою. Програми складалися з окремих модулів, що містять десяток-другий процедур і функцій. Ефективність таких програм тим вище, чим менше модулі залежать один від одного. Автономність модулів дозволяє створювати й бібліотеки модулів, щоб потім використовувати їх як будівельні блоки для програми.

Для того щоб забезпечити максимальну незалежність модулів один від одного, треба чітко відокремити ті процедури, які будуть викликатися іншими модулями – відкриті (`public`) процедури, від допоміжних, які обробляють дані, ув'язнені в цей модуль, - закритих (`private`) процедур. Перші перелічуються в окремій частині модуля – інтерфейсі (`interface`), другі беруть

участь тільки в реалізації (`implementation`) модуля. Дані, занесені в модуль, поділяються на відкриті, описані в інтерфейсі й доступні для інших модулів, і закриті, доступні тільки для процедур того самого модуля. У різних мовах програмування цей розподіл провадиться по-різному. У мові Turbo Pascal модуль спеціально поділяється на інтерфейс і реалізацію, у мові C інтерфейс виноситься в окремі "заголовні" (`header`) файли. У мові C++, крім того, для опису інтерфейсу можна скористатися абстрактними класами.

Так виникла ідея про приховання, інкапсуляцію (`incapsulation`) даних і методи їхньої обробки. Подібні ідеї періодично виникають у дизайні побутової техніки. То телевізори рясніють кнопками й стовбурчаться ручками й движками на радість допитливому телеглядачеві, панує "приладовий" стиль, то все це кудись пропадає, а на панелі залишаються тільки кнопка вмикання й ручка гучності. Допитливий телеглядач береться за викрутку.

Інкапсуляція, звичайно, провадиться не для того, щоб сховати від іншого модуля щось цікаве. Тут переслідуються дві основні цілі. Перша – забезпечити безпеку використання модуля, винести в інтерфейс, зробити загальнодоступними тільки ті методи обробки інформації, які не можуть зіпсувати або видалити вихідні дані. Друга – зменшити складність, сховавши від зовнішнього світу непотрібні деталі реалізації.

Знову виникло запитання, як саме розбити програму на модулі? Отут доречними виявилися методи вирішення старого завдання програмування – моделювання дій штучних і природних об'єктів: роботів, верстатів із програмним керуванням, безпілотних літаків, людей, тварин, рослин, систем забезпечення життєдіяльності, систем керування технологічними процесами.

Справді, кожний об'єкт – робот, автомобіль, людина – має визначені характеристики. Ними можуть служити: вага, ріст, максимальна швидкість, кут повороту, вантажопідйомність, прізвище, вік. Об'єкт може робити якісь дії: переміщатися в просторі, повертатися, піднімати, копати, рости або зменшуватися, їсти, пити, народжуватися й вмирати, змінюючи свої початкові характеристики. Зручно змоделювати об'єкт у

вигляді модуля. Його характеристики будуть даними, постійними або змінними, а дії – процедурами.

Виявилося зручним зробити й зворотне – розбити програму на модулі так, щоб вона перетворилася в сукупність взаємодіючих об'єктів. Так виникло об’єктно-орієнтоване програмування (object-oriented programming), скорочено ООП (OOP) – сучасна парадигма програмування.

У вигляді об'єктів можна подати зовсім несподівані поняття. Наприклад, вікно на екрані дисплея – це об'єкт, що має ширину (width) і висоту (height), розташування на екрані, описуване звичайно координатами лівого верхнього кута вікна, а також шрифт, яким у вікно виводиться текст, скажімо, Times New Roman, колір фону (color), кілька кнопок, лінійки прокручування й інші характеристики. Вікно може переміщатися по екрану методом move(), збільшуватися або зменшуватися в розмірах методом size(), згортуватися в ярлик методом iconify(), якось реагувати на дії миші й натискання клавіш. Це повноцінний об'єкт! Кнопки, лінійки прокручування та інші елементи вікна – це теж об'єкти зі своїми розмірами, шрифтами, переміщеннями.

Зрозуміло, вважати, що вікно само вміє виконувати дії, а ми тільки даємо йому доручення: "згорнися, розгорнися, пересунься", – це трохи несподіваний погляд на речі, але зараз можна подавати команди не тільки мишею й клавішами, але й голосом!

Ідея об’єктно-орієнтованого програмування виявилася дуже плідною й стала активно розвиватися. Виявилося зручним ставити задачі відразу у вигляді сукупності діючих об'єктів – виник об’єктно-орієнтований аналіз, ООА. Вирішили проектувати складні системи у вигляді об'єктів – з'явилося об’єктно-орієнтоване проектування (ООП) (object-oriented design, OOD).

1.2 ПРОГРАМУВАННЯ МОВОЮ С

1.2.1 Функції мови С

Будь-яка програма, написана мовою С, складається з однієї або декількох функцій. Якщо функція тільки одна, то вона обов'язково називається `main()`. Виконуюча система завжди починає виконання програми з виклику функції `main()`, потім з неї за необхідності викликаються інші функції.

Як у звичайної математичної функції, у функції мови С можуть бути аргументи. Вони перелічуються в дужках після імені функції через кому. Аргументи являють собою звичайні ідентифікатори, тому перед кожним аргументом треба записувати його тип. У функції може не бути жодного аргументу, але дужки після її імені однаково треба записувати.

Функція обчислює, як кажуть, повертає в програму, одне значення. Його тип вказується перед іменем функції. Весь вміст функції, як кажуть, її тіло, записується у фігурних дужках. Отже, структура функції така:

```
тип ім'я(аргументи){
тіло функції }
```

Якщо тип значення, що повертається, не указаний, то розуміється тип `int`. Так ми завжди записували функцію `main()`. Деякі функції не повертають ніякого значення. У такому випадку замість типу пишеться слово `void`. От приклад дуже простої функції:

```
void salve() {
printf("Вітаємо вас!\n");}
```

У функції `salve()` немає ні аргументів, ні значення, що повертається. Вона просто викликає функцію `printf()` з вітанням.

Використання функцій мови С дуже схоже на використання звичайних математичних функцій. Вони викликаються в будь-якому виразі придатого типу в тім місці, де знадобилося значення функції.

2 ОПЕРАТОР `return`

Значення, що повертається функцією, записується в операторі `return`. Його операндом служить будь-який вираз, тип якого повинен збігатися з типом значення, що повертається. Напишемо просту функцію:

```
abscmp(double a, double b) {
      return fabs(a) < fabs(b);}
```

Функція `abscmp()` порівнює абсолютні величини двох речовинних чисел `a` і `b` і повертає ціле значення – нуль або одиницю, залежно від співвідношення їхніх величин від своїх аргументів. Тип цього значення, що повертається, `int`, тому він опущений. Таку функцію зручно викликати в умовному операторі:

```
 if (!abscmp(x, 9999.99))
printf("Вихід за межі діапазону.");
```

Змінимо цю функцію так, щоб вона повертала більше по модулю значення:

```
double absmax(double a, double b) {
return fabs(a)<fabs(b) ? b : a;}
```

Тепер довелося записати тип значення, що повертається, `double`. Використовувати цю функцію можна приблизно так:

```
double x = 2.25 * absmax(a + b, sqrt(d));
```

Як прийнято в математиці, функція повертає тільки одне значення. Але цим значенням може бути покажчик на масив, рядок або більш складний об'єкт. Операторів `return` в одній функції може бути кілька. Попередню функцію `absmax ()` можна переписати так:

```
double absmax(double a, double b){
if(fabs(a) < fabs(b)) return b;
else return a; }
```

Кожний оператор `return` припиняє виконання функції. Всі наступні за ним оператори не будуть виконані. Тому оператори

return найчастіше розташовуються наприкінці функції, в умовних операторах або в операторах варіанта.

У функціях типу void, що не повертають ніякого значення, теж можуть зустрічатися оператори return, але в них не повинно бути ніякого виразу, тільки одне слово:

```
return;
```

Такий оператор return служить тільки для негайного виходу з функції.

Функція main() повертає ціле значення. Ми цим ніколи не користувалися, тому що завершення функції main() завершувало всю програму й ціле значення, повернуте функцією main(), ніде було використовувати. Насправді це значення перевіряє операційна система. Якщо воно дорівнює нулю, операційна система вважає, що функція завершилася вдало, виконавши всю свою роботу. Ненульове значення означає завершення з якимись більш-менш серйозними помилками.

Прочитавши код завершення, операційна система може залежно від його значення почати якісь дії. Найчастіше вона посилає на консоль повідомлення із вказівкою коду завершення. Програміст за значенням коду дізнається, у якому місці програми відбувся вихід з неї.

Правилом гарного тону вважається завершення функції main() оператором:

```
return 0;
```

Деякі компілятори вимагають цього, інші посилають тільки попередження про необхідність значення, що повертається, треті компілятори не вимагають нічого. Якщо ви не хочете повертати ніякого значення з функції main(), то найкраще дати їй тип void.

```
void main(){
// Тіло функції без оператора return.
}
```

При виклику функції варто звертати особливу увагу на правильний тип аргументів і значення, що повертається. Деякі компілятори стежать за відповідністю типів, посилаючи

повідомлення програмісту. Проте стандарт мови С не вимагає таких перевірок від компілятора, тому розглянемо докладніше передачу аргументів у функцію.

2.1 Передача аргументів функції

При описі функції її аргументи отримують тільки тип та ідентифікатор і тому називаються формальними аргументами. У них поки немає ніякого значення, їх можна розглядати як сховища значень, поки порожні.

При виклику функції на місці кожного аргументу записуються константи, змінні або навіть вирази відповідного типу. Це фактичні аргументи, конкретні значення, передані формальним аргументам. При виклику функції перед передачею значень формальним аргументам динамічно виділяється пам'ять, у яку й записуються значення фактичних аргументів. Пам'ять утримується функцією до закінчення її роботи й звільняється після її закінчення або після виконання оператора `return`.

Такий спосіб передачі аргументів називається передачею за значенням, оскільки передані значення фактичних аргументів копіюються в комірки, динамічно виділені для формальних аргументів. Його прекрасно описує класичний приклад функції `swap()`, що міняє місцями значення своїх аргументів:

```
void swap(int a, int b){ int t = a; a = b; b=t;}
```

Викличемо цю функцію:

```
void main(){
int x = 5, y = 7;
printf("До обміну:x=%d,y = %d\n", x, y);
swap(x, y);
printf("Після обміну: x=%d,y=%d\n",x,y);}
```

Відкомпілювавши й виконавши цю програму, ви побачите, що значення змінних `x` та `y` не змінилися! Справді, при виклику `swap(x, y)` було виділене місце в пам'яті для змінних `a` й `b`, в комірку `a` потрапило значення 5, а в комірку `b` – значення 7. Функція `swap()` чесно поміняла значення цих комірок, як показано на рисунку 2.1, але значення комірок `x` та `y`

залишилися колишніми! Функція `swap()` їх не торкалася, вона навіть не знає, де лежать фактичні аргументи.

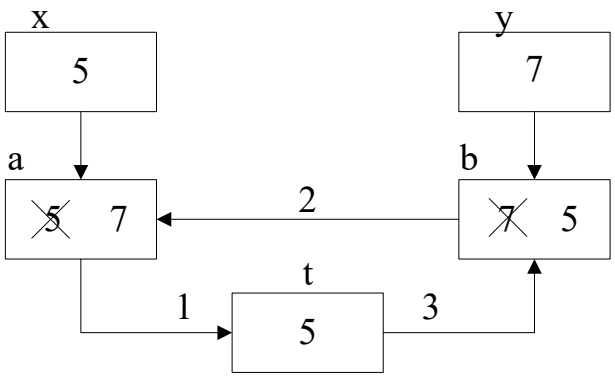


Рисунок 2.1 – Передача аргументів за значенням

Якщо ми хочемо поміняти значення `x` та `y`, то передавати треба їхні адреси, а не значення. Перепишемо для цього функцію `swap()`.

```
void swap(int *a, int *b){
int t = *a; *a = *b; *b = t;
}
```

Тепер функції передаються адреси комірок, і вона міняє значення, що перебувають за цими адресами. При виклику функції треба записувати адреси змінних:

```
void main(){
int x = 5, y -  7;
printf("До обміну: x=%d,y=%d\n",x,y);
swap(&x, &y);
printf("Після обміну: x=%d,y=%d\n",x,y); }
```

Тепер `x` та `y` помінялися місцями. Перестановки, що відбулися, показані на рисунку 2.2, де взяті умовні адреси 200 і 204.

Важливо зрозуміти, що суть справи не змінилася. Як і раніше відбувається передача за значенням, як і раніше фактичні аргументи копіюються в комірки, динамічно виділені для формальних аргументів, тільки тепер значеннями аргументів будуть адреси змінних, а не їхні величини.

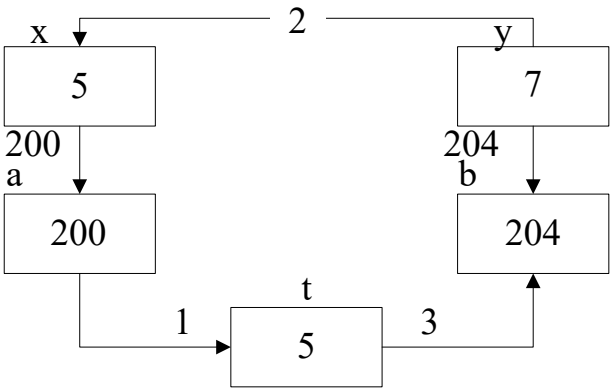


Рисунок 2.2 – Передача адрес за значенням

Вправи

1 Напишіть функцію, що повертає суму своїх аргументів.

2 Напишіть функцію, що копіює значення свого другого аргументу за адресою, указаною першим аргументом.

3 Напишіть функцію, що обчислює дискримінант квадратного рівняння.

4 Напишіть функцію, що обчислює відстань між двома точками на площині або в просторі.

3 ПЕРЕДАЧА МАСИВІВ У ФУНКЦІЮ

Оскільки ім'я масиву – це покажчик на його початок, написавши при виклику функції ім'я масиву як фактичний аргумент, ми передамо функції адресу масиву. Це найпростіший і природний спосіб передачі масиву у функцію.

У лістингу 1 написана функція `sum ()`, що підсумовує всі елементи заданого масиву. У ній передаються ім'я масиву `a` й кількість елементів `len`, які треба підсумувати. Функція `sum()` повертає суму перших `len` елементів переданого їй масиву. Потім функція `sum()` викликається у функції `main()` для конкретних масивів `x` та `y`. Наприкінці лістингу показано, як можна передати функції частину масиву.

Лістинг 1. Функція підсумовування елементів масиву

```
#include <stdio.h>
double sum(const double *a, int len){
double s = 0.0;
int k;
for (k =0; k < len; k++) s += a[k];
return s; }
void main(){
double x[] = {0.01, -3.45, 67.8, -23.5,
0.7, -23.7, 0.045};
double y[] = {2.4, 3.65, -12.2, 3.4};
double z = sum(x, 7);
printf("Сума всіх елементів масиву x=%f і
масиву в=%f\n",z, sum(y, 4));
printf( "Сума 5 елементів масиву x,
 починаючи із третього = %f\n", sum(&x[2],5));
}
```

Отже, при передачі масиву у функцію за іменем вона працює безпосередньо з комірками пам'яті масиву, змінюючи сам масив, а не його копію. Іноді функція не повинна змінювати масив, а повинна тільки читати його елементи. Така функція `sum()` у лістингу 1. Для того щоб убезпечити масив від випадкової зміни, формальний аргумент-покажчик позначений словом `const`. Після цього будь-яка спроба змінити масив буде припинена й буде зроблене повідомлення про помилку.

Вправи

1 Напишіть функцію, що знаходить суму абсолютних значень всіх елементів масиву.

2 Напишіть функцію, що обчислює діапазон значень масиву.

3 Напишіть функцію, що підраховує кількість елементів масиву, які перевищують за абсолютною величиною найбільший з перших десяти елементів масиву.

4 Напишіть функцію, що відшукує в масиві задане значення `x`. Поверніть його індекс або -1, якщо такого значення в масиві немає.

3.1 Аргументи функції `main()`

Ми завжди викликали функцію `main`() без аргументів. Дійсно, ця функція починає виконання програми, звідки ж візьмуться фактичні аргументи? Звідки викликати функцію `main`()? Давайте проаналізуємо це.

Функцію `main`() викликає одна із програм, що входять до складу операційної системи, – інтерпретатор командного рядка (`shell`). Він посилає нам запрошення, стандартний вид якого в UNIX-Подібних системах складається зі знака долара й пробілу. Він викликає програму на виконання і він же може передати їй фактичні аргументи.

Виходячи із цієї можливості, розроблювачі мови C передбачили для функції `main`() два або три аргументи. Перший аргумент цілочисельний типу `int`, другий – масив покажчиків на рядки символів типу `char**` або, еквівалентно, `char*[ ]`. Першому аргументу передається довжина цього масиву. Перший аргумент традиційно називається `argc` (`arguments count`), хоча ім'я, звичайно, може бути будь-яким, другий, як правило, називається `argv` (`arguments vector`).

Третій аргумент `arge` (arguments environment) типу `char**` використовується рідко, йому передаються масив змінних оточення і їхні значення. Отже, функцію `main`() можна записати в такий спосіб:

```
#include <stdio.h>
void main(int argc,char **argv) {int k;
for(k=0;k<argc;k++)printf("%s\n",argv[k]);}
```

Ця проста функція виводить на консоль всі свої аргументи. Нехай вона записана у файл із ім'ям `mainargs.c`. Відкомпілюємо:

```
$ gcc mainargs.c -o mainargs
```

Викликати її на виконання можна так:

```
$ ./mainargs перший, другий, третій
```

При цьому виклику значення argc буде дорівнювати 4. Аргумент argv[0] буде рівним покажчику на рядок "./mainargs", аргумент argv[1] – покажчику на рядок "перший", argv[2]=="другий", argv[3]=="третій".

Як бачите, у функцію main() в аргументі argv[0] завжди передається ім'я здійсненного файла, що містить цю функцію. Це ім'я можна використовувати для перевірки або для повторного рекурсивного виклику здійсненного файла.

Інтерпретатор командного рядка більшості операційних систем розділяє аргументи пробілами або табуляціями. Якщо треба передати аргумент, що складається з декількох слів, то їх треба взяти в лапки або апострофи. Після виклику аргумент argc буде дорівнювати 2, argv[0], як і раніше, буде містити "./mainargs", а argv[1] буде дорівнювати покажчику на рядок "перший другий третій":

```
$ ./mainargs "перший другий третій"
```

Якщо ж виклик буде виглядати так:

```
$ ./mainargs перший "другий третій",
```

значення argc буде 3, argv[0] буде містити "./mainargs", argv[1] – "перший", argv[2] – "другий третій". Таким чином, у програму завжди передаються рядки символів, навіть якщо при виклику були задані числа.

```
$ ./mainargs 123 -456 25.08
```

При такому виклику argv[1] містить "123", argv[2] – "-456", argv[3] – "25.08". Рядки символів треба перетворити в числа, як це зроблено в лістингу 2.

3.2 Функції перетворення рядків у числа

Функції перетворення рядків у числа описані у файлі stdlib.h. Функція із заголовком int atoi(char *s) (ascii to integer) повертає ціле число, отримане з рядка, на який показує покажчик s. Рядок повинний являти собою ціле значення, інакше результат буде непередбачений.

Функція із заголовком `long atol(char *s) (ascii to long)` повертає довге ціле число, отримане з рядка, на який показує покажчик `s`. Рядок повинний представляти ціле значення, інакше результат буде непередбачений.

Функція із заголовком `double atof (char *s) (ascii to float)` повертає число із плаваючою точкою подвоєної точності, отримане з рядка, на який показує покажчик `s`. Рядок повинний являти собою число, інакше результат буде непередбачений.

Лістинг 2. Перетворення рядків символів у числа

```
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[]){
if (argc < 4){
  printf("Введіть три числа\n");
  return -1;
}
int до = atoi(argv[1]),n=atoi(argv[2]);
double x = atof(argv[3]);
return 0;
}
```

Ще дві функції носять загальний характер.

Функція із заголовком `long strtol (char *s, char **ptr, int base) (string to long)` повертає довге ціле число, отримане з рядка, на який показує покажчик `s`. Покажчик `ptr` показує на перший символ рядка, що не входить у число. Число, записане в рядку `s`, вважається записаним у системі числення, основа якої передається аргументом `base`. Якщо `base==0`, то система числення визначається за записом числа: початкові `0X`, `0x` означають шістнадцяткову систему, початковий нуль – вісімкову систему.

Функція із заголовком `double strtod(char *s, char **ptr) (string to double)` повертає речовинне число подвоєної точності, отримане з рядка, на який показує покажчик `s`. Покажчик `ptr` показує на перший символ рядка `s`, що не входить у число.

4 ПРОЦЕДУРИ

Процедури, на відміну від функцій, не повертають ніякого значення. Вони обробляють значення своїх аргументів усередині себе, виводячи результати у файл, на консоль, на принтер. Процедури можуть передавати свої результати через аргументи. У цьому випадку аргументи поділяються на вхідні, передавальні вихідні значення в процедуру, і вихідні, передавальні результати роботи процедури. Вхідні фактичні аргументи можуть задаватися константами, змінними або виразами, а вихідні фактичні аргументи повинні бути тільки змінними.

Деякі аргументи можуть бути й вхідними, і вихідними. При виклику процедури вони містять вхідні значення. Потім вони змінюються усередині процедури, одержуючи вихідні значення.

Виклики процедур неможливо записати у виразах, оскільки на місці їхнього виклику не формуються ніякі значення, що повертаються. Процедури викликаються як окремі оператори – записується ім'я процедури, у дужках фактичні аргументи, після закриття дужок – точка з комою.

У мові С немає спеціальних конструкцій для процедур. Вони реалізуються функціями типу `void` або функціями типу `int`. Функцію `main()` ми завжди використовуємо як процедуру. У функціях `printf()`, `scanf()` ми ніколи не використовували значення, що повертається. Ми ніколи не записували ці функції у виразах. Функція `swap()`, написана вище, може бути класичним прикладом процедури.

Вихідні значення у функціях мови С, використовуваних як процедури, передаються за допомогою покажчиків. Так зроблено, зокрема, у функції `scanf()`, що записує результати введення за адресами, зазначеними у її аргументах.

Загалом кажучи, стиль мови С припускає використання функцій, а не процедур. Навіть у тих випадках, коли логіка обробки даних призводить до написання процедури, функції мови С повертають яке-небудь значення, наприклад, код завершення процедури, що дозволяє за необхідності відстежити хід її виконання.

4.1 Покажчики на функцію

Подібно до масивів, ім'я функції розуміється в мові С як адреса функції. Компілятор неявно створює покажчик з іменем, що збігається з іменем функції, і заносить у цей покажчик адреси початку коду функції і за ним можна відшукати функцію в оперативній пам'яті й відправити її на виконання.

Більш того, можна визначити змінну типу "покажчик на функцію, що повертає певний тип":

```
int (*f) ();
```

Не плутайте це визначення з визначенням функції, що повертає покажчик на тип `int`:

```
int *f();
```

Перед використанням покажчика на функцію треба дати значення:

```
f = sin;
```

Після цього ім'я `f` стане синонімом функції обчислення синуса `sin` і виклик `f(0.1)` буде еквівалентний виклику `sin(0.1)`.

Можливість давати синоніми відомим функціям цікава, але мало корисна. Набагато корисніше можливість передавати ім'я функції як аргумент іншої функції.

У лістингу 3 ми реалізували метод ділення навпіл для знаходження кореня нелінійного рівняння. При спробі записати цей метод у вигляді функції `bisect()` ми зіткнемося з необхідністю передати в неї функцію `f()` – ліву частину рівняння `f(x)=0`. Ось тут і буде корисним покажчик на функцію. Ми передаємо його у функцію `bisect()`. Це зроблено в лістингу 3. Потім у функції `main()` ми застосовуємо функцію `bisect()` для знаходження нулів чотирьох функцій – двох написаних відразу функцій `func1()` і `func2()` і двох стандартних тригонометричних функцій `sin()` і `cos()`.

Лістинг 3. Знаходження кореня нелінійного рівняння методом бісекції

```
#include <stdio.h>
#include <math.h>
double func1 (double x) {
  return x*x*x - 3*x*x + 3;
 }
double func2(double x){
  return x*x*x*x - 2*x*x + 1;
 }
double bisect(double a,double b,double eps,
double (*f)()){
double c;
do{
  c = 0.5 *(a + b);
  double в = f(c);
   if(fabs(y)<eps) break;
// Корінь знайдений. Виходимо із циклу.
// Якщо на кінцях відрізка [a, c]
// функція має різні знаки:
if (f(a)*y<0.0) b=c;
// виходить, корінь тут. Переносимо
// точку b у точку c.
//У противному випадку:
 else a = c;
// переносимо точку a в точку c.
//  Продовжуємо,  поки  відрізок  [a,Ь]  не
стане малий.
}while(fabs( b-a) >= eps);
return c;
}
main (){
double a = 0.0, b = 1.5, eps = 1e-5;
printf("x=%f\n", bisect(a,b,eps,func1));
printf("x=%f\n", bisect(a,b,eps,func2));
printf("x=%f\n", bisect(3.0,3.5,eps,sin));
printf("x=%f\n", bisect(1.5,2.0,eps,cos));
}
```

Вправи

1 Напишіть функцію, що обчислює інтеграл методом трапецій.

2 Напишіть функцію, що обчислює задану функцію в заданих точках і видає на екран її аргументи і значення.

3 Напишіть функцію, що повертає коефіцієнти інтерполяційного багаточлена, побудованого для заданої функції за заданою системою вузлів.

4.2 Локальні й зовнішні змінні

Формальні аргументи функції, а також змінні й масиви, визначені усередині функції, є локальними змінними. Вони відомі тільки в тій функції, у якій визначені. Інші функції нічого не знають про ці змінні й не можуть скористатися ними. Це зручно для реалізації ідеї процедурного програмування. Те саме ім'я можна використовувати в кожній функції, ніякої плутанини при цьому не буде. Програміст може створювати свої функції, записуючи в них будь-які ідентифікатори й не піклуючись про їхній випадковий збіг. У лістингу 2 формальні аргументи функцій `func1()` і `func2()` називаються однаково `x`.

Локальні змінні можна визначити не тільки на початку функції, але й у будь-якому блоці відразу після відкриття фігурних дужок. Область дії такої змінної локалізується усередині блока. Поза блоком її вже не видно й вона не може використовуватися. Її дія простирається на всі вкладені блоки, якщо тільки в них не визначена змінна з тим самим іменем.

Оскільки у функції може бути кілька рівнів вкладеності блоків, з'являються кілька рівнів визначення змінних. На кожному рівні змінну можна визначити тільки один раз. Змінні, визначені на якомусь рівні, діють у всіх вкладених блоках. Змінні, визначені у вкладених блоках, перекривають змінні з тими самими іменами, визначені раніше в зовнішніх блоках. Все це пояснює така схема:

```
/* Рівень вкладеності 0. */
int f(int x, int y){ /* Рівень 0. */
    int a = 1;  /* Рівень 1. */
```

```
    int x = 3;  /* Рівень 1. */
    if (x < y)( /* Тут x == 3 */
      int a = 2;  /* Рівень 2. */
      /* Тут a == 2. */
      static int count = 0;
      count++;
} else {
/*Рівень 2. Тут a==1. Змінна count невідома.*/
    if (x==y){ /* Тут x == 3. */
      int a = 3; /* Рівень 3. */
      /* Тут a == 3. Змінна count невідома. */
     }
/*Рівень 2. Тут a==1. Змінна count невідома.*/
    }
/*Рівень 1. Тут a == 1.Змінна count невідома.*/
}
/* Рівень 0. */
```

Деякі компілятори дозволяють визначати змінні не тільки на початку функції або блока, але й у будь-якому місці блока перед її використанням.

Локальні змінні існують тільки під час виконання блока. Комірки оперативної пам'яті їм виділяються тільки в точці їхнього визначення й звільняються при виході із блока.

Це правило можна змінити, записавши при визначенні змінної слово `static`. Так визначена змінна `count` у наведеному вище прикладі. Статичним локальним змінним пам'ять виділяється один раз відразу ж при завантаженні функції. Пам'ять зберігається до закінчення програми. Незважаючи на це статична змінна залишається локальною, вона відома тільки усередині блока.

Хоча блок, у якому визначена статична змінна, може виконуватися багато разів, початкове значення дається локальній статичній змінній тільки один раз, при початковому завантаженні. У наведеному вище прикладі змінна `count` збільшується на одиницю при кожному виконанні блока, вона підраховує, скільки разів виконується блок.

Змінні й масиви можна визначити й поза будь-якою функцією. Це зовнішні змінні, визначені на рівні 0. Вони видні у

всіх функціях, написаних нижче місця свого визначення, і можуть перекриватися локальними змінними з тими самими іменами:

```
#include <stdio.h>
int f(){
   . . .
}
#define LINES 1000
int year = 2005; /* Зовнішня змінна */
int g(){
   int a = year++;
    . . .
while (*p){
int  year=1994;/*Локальна  змінна,  у  цьому
  блоці вона перекриває зовнішню змінну. */
  . . .
  }
. . .
}
#include <math.h>
double h(){
  year++;
}
main() {
printf("%d\n", year);
}
```

Зовнішні змінні, на відміну від локальних, при визначенні обнуляються, якщо для них не задане початкове значення.

Всі функції завжди визначаються на рівні 0, мова С не допускає вкладені визначення функцій. Звичайно зовнішні змінні містять якісь глобальні значення, потрібні декільком або навіть всім функціям, що складають програму. Справа ускладнюється тим, що програму можна записати в кілька файлів. Нехай все написане вище зберігається у файлі `myprog1.c`. В іншому файлі `myprog2.c` записані ще дві функції:

```
int year; /* Неправильно. */
```

```
char *p(){
int a = year;
}
 double *q() {
        . . .
    }
```

Функції p і q теж повинні використовувати ту саму зовнішню змінну year. Оскільки файли myprog1.c і myprog2.c можуть компілюватися в різний час і в різних місцях, компілятор нічого не знає про змінну year і буде вважати помилкою її використання. Визначення змінної year, зроблене вище функції p, допоможе позбутися повідомлень компілятора, але воно створює нову змінну з таким самим іменем, а не дублює старе визначення.

Для того щоб указати компіляторові на те, що year – це не нова зовнішня змінна, а стара, визначена в іншому файлі, треба записати слово extern:

```
extern int year;

char *p(){
int a = year; }
double *q(){
  . . .
}
```

Такий запис дає не визначення, а опис зовнішньої змінної year. Побачивши опис, компілятор не стане вважати помилкою використання змінної year, але й не буде виділяти для неї комірку пам'яті.

Отже, зовнішня змінна повинна бути визначена в одному файлі, при визначенні для неї резервується комірка оперативної пам'яті. У всіх інших файлах ця змінна повинна бути описана словом extern.

Словом extern можна позначити й локальну змінну. Ця позначка теж буде означати, що змінна визначена в іншому файлі, але вона залишається локальною, область її видимості обмежується блоком.

Зовнішню змінну можна визначити зі словом `static`. Зміст цього слова для зовнішніх змінних зовсім інший, ніж для локальних змінних. Область видимості статичної зовнішньої змінної обмежується залишком файла, у якому вона визначена. В інших файлах статична зовнішня змінна невідома, у них ми можемо визначити іншу зовнішню змінну з тим самим ім'ям.

5 ПРОТОТИПИ ФУНКЦІЙ

При записі програми, що складається з декількох функцій, можливо, рознесених по декількох файлах, виникає ще одна складність. Якась функція може бути викликана раніше, ніж вона записана у файлі. Більш того, опис функції може виявитися в іншому файлі, недоступному компіляторові. У цих випадках компілятор не зможе перевірити відповідність типів формальних і фактичних аргументів. Він навіть не зможе перевірити збіг кількості аргументів. Відсутність перевірок призведе до появи безлічі помилок, які буде дуже важко виявити й виправити.

Для того щоб уникнути такого роду помилок, у мову С введене правило: якщо функція описана пізніше її виклику, то до опису функції, у якій вона викликається, варто написати прототип викликуваної функції. Прототип функції – це її заголовок, що включає тип значення, що повертається, ім'я функції й список аргументів у дужках. Відразу після закриття дужок ставиться крапка з комою:

```
#include <stdio.h>
double sum(const double *a, const int len);
void main () {
     double           x[]={0.01,-3.45,67.8,-
       23.5,0.7,-23.7,0.045};
  double z = sum(x, 7);
}
double sum(const double *a, const int len){
  double s = 0.0; int k;
  for (k =0; k < len; k++) s += a[k];
  return s;
}
```

Прототип містить всі відомості, необхідні компілятору для правильного виклику функції. Правила гарного тону пропонують записувати на початку файла прототипи всіх функцій, наявних у ньому. Це полегшує читання програми. Відкривши файл, ви відразу бачите, які функції він містить, і можете зрозуміти за їхніми іменами і набором аргументів, що роблять ці функції.

Прототипи функцій поряд з визначеннями констант часто записують в окремий файл і підключають його в початок потрібного файла директивою препроцесора `#include`.

6 РЕКУРСІЯ

У математиці часто зустрічається рекурсивне визначення функції. Значення функції від аргументу `n` визначається через одне або кілька попередніх значень `n-1, n-2, ...` . Класичний приклад – визначення факторіала: `0!=1, n!=n(n-1)!`.

За цим визначенням `4!=4•3!`, але `3!=3•2!`, а `2!=2•1!`, `1!=1•0!` і ми дійшли до кінця рекурсії, розкрили вкладені одне в одного визначення. Після цього рухаємося у зворотному порядку, виконуючи множення: `1!=1•1=1, 2!=2•1=2, 3!=3•2=6`, нарешті, `4!=4•6=24`.

Більш складна послідовність чисел Фібоначчі `1, 1, 2, 3, 5, 8, ...` записується рекурсивною формулою: $a_1=a_2=1$, $a_n=a_{n-1}+a_{n-2}$.

Мова С припускає виклик функції усередині самої функції, тому рекурсію легко запрограмувати:

```
int fact(int n) {
if (n<0) return -1; /* Неправильний аргумент. */
if (n==0) return 1;
return n*fact(n-1);
}
```

Обчислення n-го числа Фібоначчі нітрохи не складніше:

```
int fib(int n){
  if (n<=0) return -1;
  if (n==1 || n==2) return 1;
  return fib(n-1)+fib(n-2);
}
```

Звичайно, обчислювати рекурсивно такі прості функції не варто, рекурсія вимагає великої витрати ресурсів. Тут досить простого циклу:

```
int fact(int n){
 if (n<0) return -1; /* Помилка. */
 int k,f=1;
 for (k=2; k<=n; k++) f*=k;
 return f;
}
```

Числа Фібоначчі теж краще обчислювати в циклі:

```
int fib(int n){
   if (n<=0) return -1;
   int k,a1=1,a2=1,f=1;
   for (k=3; k<=n; k++){
     f=a1+a2;
     a1=a2;
     a2=f
   }
   return f;
}
```

У більш складних випадках, наприклад при побудові дерев, рекурсія допомагає написати чітку й коротку програму.

Створюючи рекурсивну функцію, варто ретельно простежити за тим, щоб рекурсія не виявилася нескінченною, щоб вкладення розкрилися до кінця й досягли конкретних початкових значень. Помилки, викликані нескінченною рекурсією, виявляються тільки під час виконання програми, коли їх важко відстежити й виправити.

7 ШВИДКЕ СОРТУВАННЯ

Найшвидший метод сортування, так зване швидке сортування (quick sort), заснований на рекурсії.

Метод швидкого сортування полягає в тім, що вибирається середній елемент масиву, наприклад, елемент `a[LEN/2]`, де `len` – довжина масиву. Ліворуч від цього елемента вибираються елементи, більші `a[LEN/2]`, а праворуч – елементи, менші `a[LEN/2]`, після чого відбувається перестановка цих елементів. У результаті лівіше від обраного елемента `a[LEN/2]` опиняться всі елементи, менші від нього, а правіше – всі елементи, більші від нього. Після цього алгоритм рекурсивно повторюється окремо для лівої половини масиву й для правої половини.

Технічно це оформлено у вигляді рекурсивної функції `quicksort(left, right)`, у яку передаються індекси лівого `left` і правого `right` кінця підмасиву. У функції встановлюються індекси `l` і `r`, які пересуваються від кінців до середини масиву ліворуч і праворуч. Індекс `l` зупиняється на першому елементі масиву, більшому, ніж середній елемент `m`, а індекс `r` – на першому праворуч елементі масиву, меншому, ніж `m`. Після цього елементи `a[l]` і `a[r]`, якщо такі елементи знайдені, міняються місцями, а індекси `l` і `r` продовжують просуватися до середини масиву.

Після закінчення цього циклу виявляється, що `l>r`. Якщо після цього `left<r` або `l<right`, то виклик функції треба повторити з новими аргументами (лістинг 4).

Лістинг 4. Швидке сортування масиву

```
    #include <stdio.h>
    #define LEN 10 int a[] = {15, 12, 5, 7, 0,
-2, 4, 8, -3, 10};
    void quicksort(int left, int right){
    int m = a[(left + right)/2];
    int l=left,r=right;
    while (l<=r){
      while (a[l]<m) l++;
      while (a[r]>m) r--;
```

```
   if (l<=r){
     int t=a[l]; a[l]=a[r]; a[r]=t;
     l++; r--;
    }
  }
  if (left<r) quicksort(left,r);
  if (l<right) guicksort(l,right);
  }
main(){
quicksort(0, LEN-1);
}
```

Контрольні питання

1 Що таке парадигма програмування?

2 Що таке процедурне програмування?

3 Що таке модульне програмування?

4 Яка різниця між процедурою й функцією?

5 Чи можна написати процедуру в мові С?

6 Як повернути два або більше значень з функції?

7 Чи можна описати функцію усередині іншої функції?

8 Чи можна створити зовнішню змінну усередині функції?

9 Чи можна записати текст однієї функції в кілька файлів?

10 Чи обов'язково записувати прототип для кожної функції мови С?

11 У яких випадках прототип необхідний?

12 Рекурсія називається взаємною, якщо дві функції викликають одна одну. Чи можна в мові С створити взаємно рекурсивні функції?

8 ФУНКЦІЯ ВИВЕДЕННЯ `printf()`

Виведення значень різних виразів за допомогою функції `printf()` відбувається таким чином. Самі вирази записуються другим, третім і так далі параметром через кому, а в першому параметрі – покажчикові на рядок – спеціальними символьними виразами (специфікаціями) описується формат виведення значення кожного виразу. Це виглядає приблизно так:

```
#include <stdio.h>
char c = '?' ;
int k = 15;
printf("c=%c,код c=%d, k=%d,k+c=%6d\n",c,c,
k,k+c);
```

Специфікація починається зі знака відсотка й закінчується однією з наступних букв, що показують вид подання даного, сформованого при виведенні:

`c (character)` – один символ;

`d,i (decimal, integer)` – ціле число в десятковій системі числення;

`u (unsigned)` - ціле число без знака в десятковій системі числення;

`o (octal)` – ціле число без знака у вісімковій системі числення;

`x, X (hexadecimal)` – ціле число без знака в шістнадцятковій системі числення. Якщо зазначено параметр `x`, то шістнадцяткове число буде записано малими буквами, наприклад `2f34cd`, якщо X, то виведене число буде записано великими буквами: `2F34cd;`

`f (float)` - речовинне число буде виведене з фіксованою точкою із шістьма цифрами в дробовій частині;

`e, E (exponent)` – речовинне число буде виведене із плаваючою точкою із шістьма цифрами в дробовій частині мантиси. Число нормалізується так, що в цілій частині виводиться одна ненульова цифра. Якщо зазначено параметр `e`, то при виведенні буде записана мала буква, наприклад `0.12345e-02;` якщо `E`, то одержимо `0.12345 E-02;`

`g, G (global)` - буде виведено шість значущих цифр речовинного числа з фіксованою або із плаваючою точкою; якщо його порядок менше `-4` або більше його точності;

`s (string)` - рядок символів, що закінчується нуль-символом.

У написаному вище прикладі чотири специфікації - `%c, %d, %d, %6d.` Вони застосовуються до виведених даних у тім порядку, у якому записані в першому аргументі функції `printf().` Перша специфікація `%c` застосовується до

символьної змінної c і формує замість себе знак питання, друга специфікація %d застосовується до наступного аргументу c і формує десятковий код 63 знака питання. Третя специфікація %d застосовується до цілої змінної k і формує десяткове число 15, а четверта – %6d – до останнього аргументу – виразу k + c. Вона дає число 78 у десятковій системі числення.

Якщо число специфікацій менше числа аргументів, то останні аргументи не виводяться. Якщо їхнє число більше числа аргументів, то зайві специфікації виведуть сміття, що лежить у наступних комірках оперативної пам'яті.

На місці специфікацій у виведеному рядку з'являються сформовані ними значення, причому пробіли між значеннями не формуються автоматично, їх треба вказувати спеціально. Інші символи виводяться так, як записані в першому аргументі функції printf(). Отже, наведена вище програма дасть таке виведення:

```
c = ?, код c = 63, k = 15, k + c =     78.
```

Остання специфікація %6d трохи відрізняється від інших. Вона містить число 6. Це число позицій на консолі, виділюваних для виведеного значення. Якщо значення коротше, скажімо, у нашім прикладі число 78 займає тільки дві позиції, то воно зрушується вправо, ліворуч залишаються пробіли. Якщо записати в числовій специфікації нуль, наприклад %06d, то замість пробілів будуть виведені нулі, ми одержимо 000078. Якщо записати дефіс, %-6d, то значення буде зрушуватися вліво, у нашім прикладі пробіли залишаться після числа 78.

Число позицій можна вказати не тільки за допомогою %d, але й у будь-якій іншій специфікації. Воно задає мінімальну ширину поля, у яке поміщається виведене значення. Якщо виведене значення вимагає більше позицій, ніж зазначено в специфікації, то поле розширюється до необхідних розмірів.

Після числа позицій через точку можна записати ще одне число, причому перше число можна й не записувати. Для цілих значень число після точки буде означати мінімальну кількість виведених цифр, наприклад %10.5d, %4.3u, %.2o,

%6.6x, %.6x. Якщо значення коротше, то ліворуч будуть виведені незначущі нулі.

Для речовинних специфікацій (%f, %e, %E) число після точки означає кількість виведених цифр у дробовій частині, наприклад %5.2f, %io.5e, %.5E. Для специфікації %g, %g воно означає кількість виведених значущих цифр, наприклад %8.2g, %.8G.

Якщо число після точки зазначено в специфікації %s, то виводяться тільки перші символи рядка, наприклад при виведенні printf("%.5s\n", "Перші символи"); одержимо тільки слово Перші.

У чисел зі знаком звичайно виводиться тільки мінус, плюс не виводиться. Якщо в специфікації поставити плюс, наприклад, %+8d, то числа будуть показані зі своїм знаком. Для нечислових значень плюс ігнорується. Якщо ж специфікацію почати із пробілу, % d, то перед позитивним числом не ставиться плюс, але залишиться пробіл, що зручно для вирівнювання стовпців у таблиці чисел.

Для виведення довгих цілих значень типу long перед буквами d, i, o, i, x, X, що завершують специфікацію, треба записати малу букву l, інакше функція printf() візьме тільки перші байти довгого значення й сформує з них виведене число. Для виведення значень типу double перед буквами f, e, E, g, G треба поставити малу букву l, інакше функція printf() візьме тільки перші байти речовинного числа з подвоєною точністю. Для виведення значень типу long double перед буквами f, e, E, g, G треба поставити велику букву L, інакше функція printf() візьме тільки перші байти.

Нарешті, якщо на початку специфікаці поставити знак номер (#), наприклад %#o, %#5x, %#7.2f, %#e, %#g, то вісімкові числа будуть виводитися з початковим нулем, ненульові шістнадцяткові числа будут починатися з 0x або 0X, у речовинних числах буде ставитися крапка після цілої частини, навіть якщо в дробовій частині нема цифр. Специфікації %g, %G, крім того, виводять незначущі хвостові нулі.

Отже, знак відсотка в першому аргументі функції printf() має особливе значення. Він означає початок

специфікації. Після нього компілятор буде шукати одну з букв, що завершують специфікацію. Якщо її немає, то результат важко вгадати. Якщо ж у тексті зустрічається знак відсотка, то його треба подвоїти.

Наприклад,

```
printf("Якщо ratio < 80%%, то збільште кеш\n").
```

Вправи

1 Напишіть просту програму, що містить змінні різних типів і функцію `printf()` із різними специфікаціями, що виводить значення цих змінних і виразів із ними. Зверніть увагу на отримане виведення, особливо для значень типу `long`, `double` і `long double`.

2 Спробуйте записати у функції `printf()` підряд кілька аргументів, що містять вирази `++k`, і переконайтеся, що вони не обов'язково обчислюються зліва направо.

9 ФУНКЦІЯ ВВЕДЕННЯ `scanf()`

Тепер навчимося вводити вихідні дані із клавіатури. У стандартній бібліотеці функцій введення-виведення, описаній у файлі `stdio.h`, є кілька функцій, призначених для введення вихідних даних. Ми їх вивчимо поступово, а поки будемо користуватися функцією форматованого введення `scanf()`.

Чому треба форматувати введення? З погляду комп'ютера, все, що ви вводите із клавіатури, – це рядок символів. Навіть коли ви набираєте число `12345`, програма одержує після декількох перетворень п'ять байтів з кодами натиснутих клавіш, у цьому прикладі `0011000100110010001100110011010000110101`. Ці п'ять байтів треба перетворити в чотири байти, що зберігають ціле число типу `int`. Цим й займається функція `scanf()`.

Функція `scanf()` аналогічна функції `printf()`. Її перший аргумент теж покажчик на рядок формату, що містить такі самі специфікації, але наступні аргументи – не змінні або вирази, а адреси змінних, у які будуть записані відформатовані

вихідні дані. Про адреси ми будемо говорити пізніше, а поки для того щоб одержати адресу змінної, просто ставте перед нею амперсанд.

Наприклад,

```
main ()
{
int a, k = a;
float x;
scanf("%d%d%f", &a, &k, &x);
 }
```

Як бачите, у рядку формату між специфікаціями немає ніяких роздільників. У такому випадку вводяться дані, що треба розділяти пробілами. Функція почне виконуватися й читати набрані на клавіатурі символи після натискання клавіші <Enter>.

Будь-який символ, записаний між специфікаціями, вважається роздільником і повинен вводитися із клавіатури. Наприклад, записана функція

```
scanf(“%d,%d,%f”, &a, &k, &x);
```

У цьому випадку дані, що вводяться, треба розділяти комами й необов'язковими пробілами. Розділяти дані можна, регулюючи ширину поля, виділеного для них. Нехай, наприклад, написана функція

```
scanf (“%2d%3d%5.2f”, &a, &k, &x);
```

Ми набрали цифри 1234567890. Після цього змінна a одержить значення 12, у змінну k буде записане значення 345, а змінна x одержить значення 678.9.

Так само, як і у функції printf(), при введенні значень довгих змінних типу long і double перед буквою відповідної специфікації треба поставити букву l, а при введенні значень типу long double – букву L. Деякі складності виникають при введенні символьних даних. Адже будь-який роздільник – це символ, що буде вважатися значенням, що вводиться. У даному прикладі набирати символи треба підряд без будь-яких роздільників:

```
char ch1, ch2, ch3;
scanf("%c%c%c", &ch1, &ch2, &ch3);
```

Якщо записати наступну функцію, то будь-який роздільник цілих значень потрапить у символьну змінну `ch1`:

```
scanf("%d%c%d', &k &ch1, &k);
```

Вправи

1 Напишіть просту програму, що містить змінні різних типів, і функцію `scanf()` із різними специфікаціями, що заносить значення в ці змінні. Виведіть значення змінних функцією `printf()` і подивіться уважно на отримане виведення. Переконайтеся в тім, що специфікації точно відповідають типу змінної.

10 ЗНАХОДЖЕННЯ КОРЕНІВ КВАДРАТНОГО РІВНЯННЯ

Озброївшись функціями `scanf()` і `printf()`, наведемо приклад програми, що містить численні перевірки. Обчислення коренів квадратного рівняння $ax^2+bx+c=0$ для будь-яких коефіцієнтів, у тому числі й нульових реалізується блок-схемою, зображеною на рисунку 10.1.

У цій програмі використана функція обчислення модуля `fabs()` речовинного числа й функція добування квадратного кореня `sqrt()` з речовинного числа. Це функції зі стандартної бібліотеки математичних функцій мови С. Оскільки всі обчислення з речовинними числами провадяться приблизно, ми вважаємо, що коефіцієнт рівняння дорівнює нулю, якщо його модуль менше `0.00000001`. Зверніть увагу на те, як операція присвоювання при обчисленні дискримінанта вкладена в порівняння з нулем.

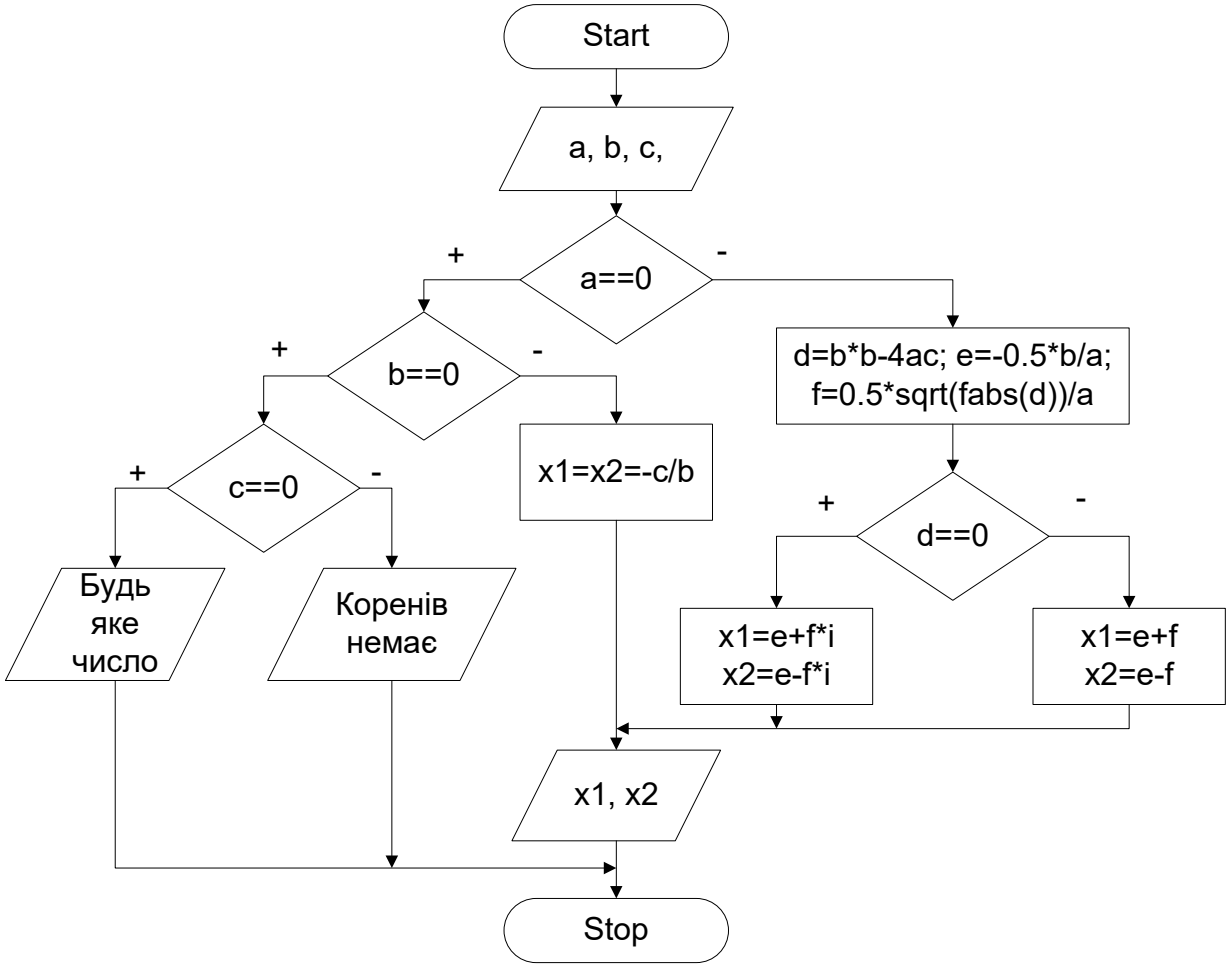


Рисунок 1.10 – Блок-схема знаходження коренів квадратного рівняння

Лістинг програми обчислення кореня квадратного рівняння

```
#include <stdio.h>
#include <math.h>
main()
{
float a=0.5,b=-2.7,c=3.5,d,eps=1e-5;
printf("Введіть точність обчислень:");
 scanf("%f", &eps);
printf("\nвведіть коефіцієнти рівняння через
пробіли: ");
 scanf("%f%f%f", &a, &b, &c);
 if (fabs(a)<eps)
  if (fabs(b)<eps)
   if (fabs(c)<eps)
printf("\nрішення - будь-яке число.\n");
```

```
   else printf (\nрішень немає.\n");
  else printf("x1=x2=%f\n", -c/b);
 else // Коефіцієнти не дорівнюють нулю
if ((d = b*b - 4*a*c) < 0.0)
{// Комплексні корені
 d = 0.5*sqrt(-d)/a;
 a = -0.5*b/a;
printf("\nx1=%f+i%f,x2=%f-i%f\n",a,d,a,d);
}
else
{// Речовинні корені
 d = 0.5*sqrt(d)/a;
 a = -0.5*b/a;
printf("\nx1=%f, x2=%f\n",a+d,a-d);
}
}
```

11 ОПЕРАТОРИ ЦИКЛУ

Основний оператор циклу – `while` – у мові С виглядає надзвичайно лаконічно:

```
while (вираз) оператор
```

Дужки, що оточують вираз, обов'язкові, пробіли, як ми вже говорили, необов'язкові, тому що дужки – це теж роздільники.

Оператор працює так. Спочатку обчислюється вираз в дужках. Якщо його значення не дорівнює нулю (вираз істинний – `true`), то виконується оператор, що утворить цикл. Потім знову обчислюється вираз й діє оператор, і так доти, поки вираз не стане рівним нулю (стане помилковим – `false`).

Якщо вираз із самого початку дорівнює нулю (`false`), то оператор не буде виконаний ніколи.

Таким чином, оператор `while` реалізує блок-схему на рисунку 11.1.

Попередня перевірка забезпечує безпеку виконання циклу, дозволяє уникнути переповнення, ділення на нуль та інших

неприємностей. Тому оператор `while` є основним, а в деяких мовах програмування і єдиним оператором циклу.

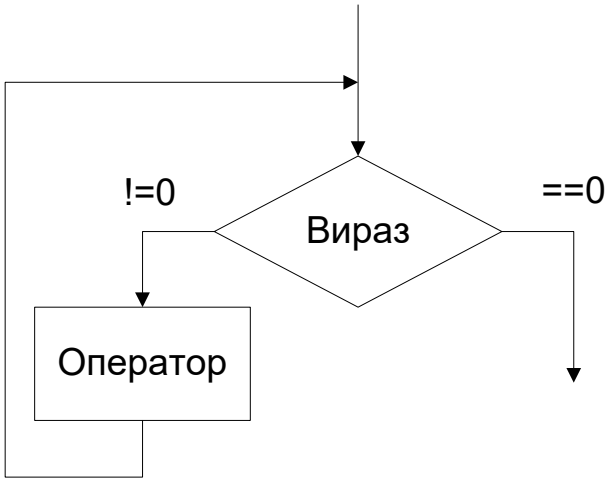


Рисунок 11.1 – Дія оператора циклу `while`

Оператор у циклі може бути й порожнім, наприклад:

```
int i = 0;
double s = 0.0;
while ((s+=1.0/++i)<10);
```

Цей фрагмент коду обчислює кількість додавань `i`, які необхідно зробити, щоб гармонійна сума `s` досягла значення 10. Такий стиль характерний для мови С. Не варто ним захоплюватися, щоб не перетворити текст програми в шифровку, на яку ви самі через пару тижнів будете дивитися зі здивуванням.

Можна організувати й нескінченний цикл – `while(1) оператор`. Звичайно, з такого циклу варто передбачити якийсь вихід, наприклад, оператором `break`. В іншому випадку програма зациклиться й вам доведеться припиняти її виконання "комбінацією із трьох пальців" `<Ctrl>+<Alt>+<Del>` в Windows 95/98/ME, комбінацією `<Ctrl>+<C>` в UNIX або за допомогою Task Manager в Windows NT/2000/XP/2003.

Якщо в цикл треба включити декілька операторів, то варто утворити блок операторів `{ }`.

Другий оператор циклу – `do-while` – має такий вигляд:

```
do оператор while (вираз);
```

Тут спочатку виконується оператор, а потім відбувається обчислення виразу. Цикл виконується, поки вираз залишається не рівним нулю (правильним). Блок-схема цих дій показана на рисунку 11.2.

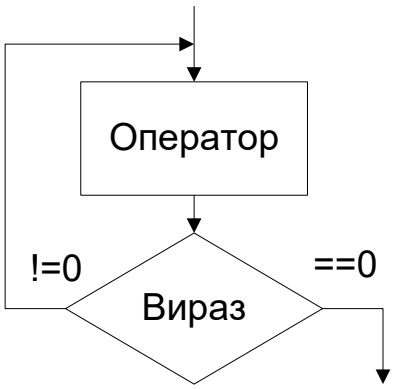


Рисунок 11.2 – Дія оператора циклу `do-while`

Істотна різниця в організації циклів цими двома способами в тому, що в останньому оператор обов'язково виконається хоча б один раз. Наприклад, нехай задана якась безперервна функція `f(x)`, що має на відрізку `[a,b]` рівно один корінь. У лістингу 5 наведена програма, що обчислює цей корінь приблизно методом ділення навпіл. Цей самий метод часто називають методом бісекції, використовуючи англійський термін, або, по-грецьки, методом дихотомії.

Суть методу в тому, що заданий відрізок ділиться навпіл і з'ясовується, у якій половині знаходиться корінь. Це можна зробити перевіркою значень функції на кінцях кожної половини. Там, де лежить корінь, значення безперервної функції мають різні знаки. Після цього обрана половина відрізка знову ділиться навпіл, і все триває доти, поки не буде досягнута потрібна точність.

Оскільки ділення навпіл треба зробити хоча б один раз, у лістингу 5 обраний цикл `do-while`.

Лістинг 5. Знаходження кореня нелінійного рівняння методом бісекції

```
#include <stdio.h>
double f(double x) {
```

```
    return      x*x*x-3*x*x+3;//Якщо      хочете,
напишіть    // що-небудь інше...
     }
    main(){
     double a =0.0, b=1.5, c, y, eps = 1e-8;
     do{
     c = 0.5*(a + b);
     в = f(c);
     if (fabs(y) < eps) break;
     // Корінь знайдений. Виходимо із циклу.
     // Якщо на кінцях відрізка [a, c]
     // функція має різні знаки:
     if (f(a)*y < 0.0) b = c;
     //виходить, корінь тут. Переносимо
     // точку b у точку c.
     // В іншому випадку:
     else a = c;
     // переносимо точку a в точку c.
    //  Продовжуємо, поки відрізок [a, b] не
стане малий.
     }while(fabs( b-a) >= eps);
     printf("x = %f, f(%f) = %f\n", c, c, y);
     }
```

Ця програма складніше від попередніх прикладів: у ній, крім функції `main()`, є ще функція `f()`. Ця функція дуже проста: вона обчислює значення багаточлена й повертає його як значення функції, причому все це виконується одним оператором:

```
return виразу;
```

У функції `main()` з'явився ще один новий оператор `break`, що просто припиняє виконання циклу, якщо ми завдяки щасливому випадку наткнулися на наближене значення кореня.

Вправи

1 Напишіть програму, що підраховує кількість цифр заданого цілого числа.

2 Напишіть програму, що дає суму цифр заданого цілого

числа.

3 Напишіть програму, що переставляє цифри заданого цілого числа у зворотному порядку. Ви можете скористатися алгоритмом на рисунку 11.3.

4 Напишіть програму, що підраховує суму величин, зворотних квадратам перших натуральних чисел $S=1+1/2^2+1/3^2 + 1/4^2...$, із заданою точністю.

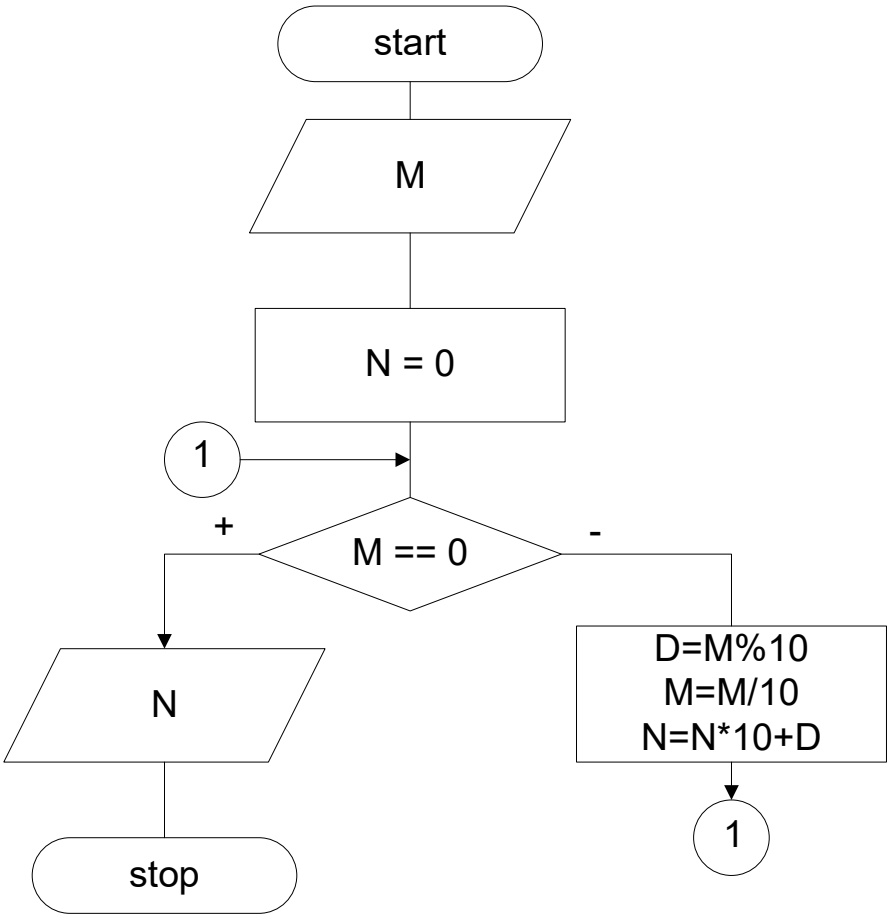


Рисунок 11.3 – Перестановка цифр цілого числа

Наведемо ще один повний приклад типового обчислення. Відомо, що точність обчислення суми перших членів збіжного знакозмінного ряду можна оцінити абсолютною величиною останнього відкинутого члена. Застосуємо цю теорему для обчислення синуса заданого числа із заданою точністю `eps` за відомою формулою:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{(-1)^n x^{2n+1}}{(2n+1)!} + \ldots$$

Ці обчислення проводяться в програмі лістингу 6. Вони порівнюються зі значенням синуса того самого аргументу, обчисленим функцією `sin(x)` зі стандартної бібліотеки мови С.

Зверніть увагу на те, що зміна знака доданків забезпечується тим, що ми просто ставимо знак мінус при обчисленні кожного наступного доданка p. Не треба обчислювати ніякий степінь мінус одиниці.

Друга особливість програми полягає в тому, що ми не обчислюємо повністю наступний доданок `p`, а одержуємо його з попереднього доданка множенням на квадрат аргументу `x` і діленням на добуток наступних цілих чисел `k*(k+1)`. Це викликано тим, що аргумент `x` може бути досить великим за абсолютною величиною й зведення його в більший ступінь швидко викличе переповнення. Це ж неминуче трапиться при обчисленні факторіала. Ми ж обчислюємо відношення порівняно невеликих величин, що не повинне викликати переповнення.

Лістинг 6. Обчислення синуса за допомогою відрізка ряду Тейлора

```
#include <stdio.h>
main{
float eps=1e-8, x=0.0;
printf("Введіть точність обчислень: ") ;
scanf("% f", &eps);
printf("Введіть аргумент синуса: ");
scanf("%f", &x);
double s=0.0, p=x;
int k=2;
while (fabs(p)>=eps)
{
S += p;
p = -p * x * x / (k * (k + 1) ) ;
k += 2;
}
printf("x=%f, s=%f, sin=%f"\n",x,s,sin(x));
}
```

Третій оператор циклу – `for` – виглядає так:

```
for (списоквир1; вираз; списоквир2) оператор
```

Перед виконанням циклу обчислюється список виразів `списоквир1`. Це нуль або кілька виразів, перерахованих через кому, тобто операцію "кома". Вони обчислюються зліва направо, і в наступному виразі вже можна використовувати результат попереднього виразу. Як правило, тут задаються початкові значення змінним циклу.

Потім обчислюється другий вираз. Якщо він відмінний від нуля, то діє оператор, потім обчислюються зліва направо вираз зі списку виразів `списоквир2`. Далі знову перевіряється вираз. Якщо він не дорівнює нулю, то виконується `оператор` і `списоквир2` і т. д. Як тільки вираз стане рівним нулю, виконання циклу закінчується. Коротше кажучи, виконується послідовність операторів: `списоквир1`;

```
while (вираз){
оператор
списоквир2; }
```

з тим виключенням, що якщо `оператор` у циклі є оператором `continue`, то `списоквир2` все-таки виконується.

Багато компіляторів допускають зробити в `списоквир1` одне визначення змінних обов'язково з початковим значенням. Такі змінні відомі тільки в межах цього циклу.

Будь-яка частина оператора `for` може бути відсутньою: цикл може бути порожнім, вираз в заголовку теж, при цьому крапки з комами зберігаються. Можна задати нескінченний цикл: `for( ; ; ) оператор`. У цьому випадку в тілі циклу варто передбачити який-небудь вихід.

Хоча в операторі `for` закладені більші можливості, використовується він, головним чином, для перерахувань, коли їхнє число відомо, наприклад:

```
int s = 0, k;
for (k = 1; k <= N; k++) s += k * k;
```

Цей фрагмент коду обчислює суму квадратів перших `n` натуральних чисел. Подивіться, якими різними способами можна записати ці оператори.

```
int s = 0, k = 1;
for (  ; k <= N; k++) s += k * k;
```

Тут всі початкові дії винесені з оператора циклу. На їхньому місці в заголовку циклу залишилася тільки точка з комою.

```
int s = 0, k = 1;
for (  ; k <= N;) s += k * (k++);
```

Тут зміна змінної `k` винесена в цикл, у заголовку залишилася тільки умова. Можна зробити навпаки й внести обчислення в заголовок:

```
int s = 0, k = 1;
for (  ; k <= N; s += k * (k++));
```

Можна скористатися операцією "`кома`" і написати так:

```
int s = 0, k = 1;
for (  ; k <= N; s += k * k, k++});
```

Нарешті, можна всі дії внести в заголовок циклу:

```
int s,k; for(s=0,k=1;k<=N;s+=k*k,k++));
```

Не захоплюйтеся такими незвичайними конструкціями. Літакобудівники говорять: "Красивий літак краще літає". У програмуванні діє таке саме недоведене містичне правило: чим простіше й красивіше програма, тим краще вона працює. Намагайтеся записувати оператори в такому самому вигляді, як це робили творці компілятора. Будьте впевнені: вони перевірили свої записи сотні разів і довели їх до досконалості!

Вправи

5 Напишіть повну програму, у якій перевірте дію всіх способів запису цього циклу.

6 Напишіть програму, що підраховує суму величин, зворотних квадратам перших ста натуральних чисел.

7 Число $\frac{\pi}{4}$ можна обчислити як суму знакозмінного ряду $1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\dots$, але вона дуже повільно сходиться. Яку точність дасть сума першої тисячі складових цього ряду?

Оператор `continue` використовується тільки в операторах циклу. Він складається тільки з одного слова й здійснює

негайний перехід до наступної ітерації циклу. У наступному фрагменті коду даний оператор дозволяє обійти ділення на нуль:

```
 int  i, j = N / 2;
for (i = 0; i < N; i++)
 {
if (i == j) continue;
s += 1.0/(i-j);
```

Коли значення змінної `i`, зростаючи, зрівняється зі значенням змінної `j`, відповідний доданок не буде обчислюватися й додаватися до `s`, а відразу ж відбудеться перехід до наступної ітерації, у якій `i` уже буде більше, ніж `j`.

12 ОПЕРАТОР `break`

Оператор `break` використовується в операторах циклу й операторі варіанта для негайного виходу із цих операторів. Це зручно, наприклад, при пошуку в циклі заданого значення. Як тільки воно буде знайдено, потрібно відразу припинити виконання циклу й вийти з нього.

Оператор `break`, як і оператор `continue`, складається лише з одного слова. Він використовується найчастіше в умовних операторах. Наприклад:

```
double s = 0.0; int k = 1;
while(1){
s += 1.0 / k++;
if (s > 10.0) break; }
 printf("k = %d\n", k);
```

13 ОПЕРАТОР ВАРІАНТА `switch`

Оператор варіанта `switch` реалізує розгалуження на кілька гілок. Кожна гілка відмічається константою або константним виразом якого-небудь цілого типу й вибирається, якщо значення

визначеного виразу збіжиться із цією константою. Вся конструкція виглядає так:

```
switch (Цілий вираз)
{
case конствир1: Оператор1
case конствир2: Оператор2
. . . .
case конствир: ОператорN
default: ОператорDef
}
```

Цілий вираз, що стоїть у дужках, може бути типу `char, short, int, long`. Оператор варіанта виконується так. Всі константні вирази обчислюються заздалегідь, на етапі компіляції, і повинні мати відмінні одне від одного значення. Спочатку обчислюється цілочисельний `цілий вираз`. Якщо він збігається з однією з констант, то виконується оператор, відзначений цією константою. Потім виконуються `(fall through labels)` всі наступні оператори, включаючи й `операторDef,` і виконання оператора варіанта закінчується.

Якщо ж жодна константа не дорівнює значенню виразу, то виконується `операторDef` і всі наступні за ним `оператори`. Тому гілка `default` повинна записуватися останнім рядком оператора `switch`, хоча синтаксис не забороняє їй розташовуватися й між іншими варіантами. Гілка `default` може не існувати, у цій ситуації оператор варіанта взагалі нічого не робить. Схематично робота оператора `switch` показана на рисунку 13.1.

Таким чином, константи у варіантах `case` відіграють роль тільки міток, точок входу в оператор варіанта, а далі виконуються всі оператори, що залишилися, у порядку їхнього запису. Найчастіше необхідно виконати тільки одну гілку операторів. У такому випадку використовується оператор `break`, що відразу ж припиняє виконання оператора `switch`. З іншого боку, може знадобитися виконати той самий оператор у різних гілках `case`. У цьому випадку ставляться кілька міток `case` підряд.

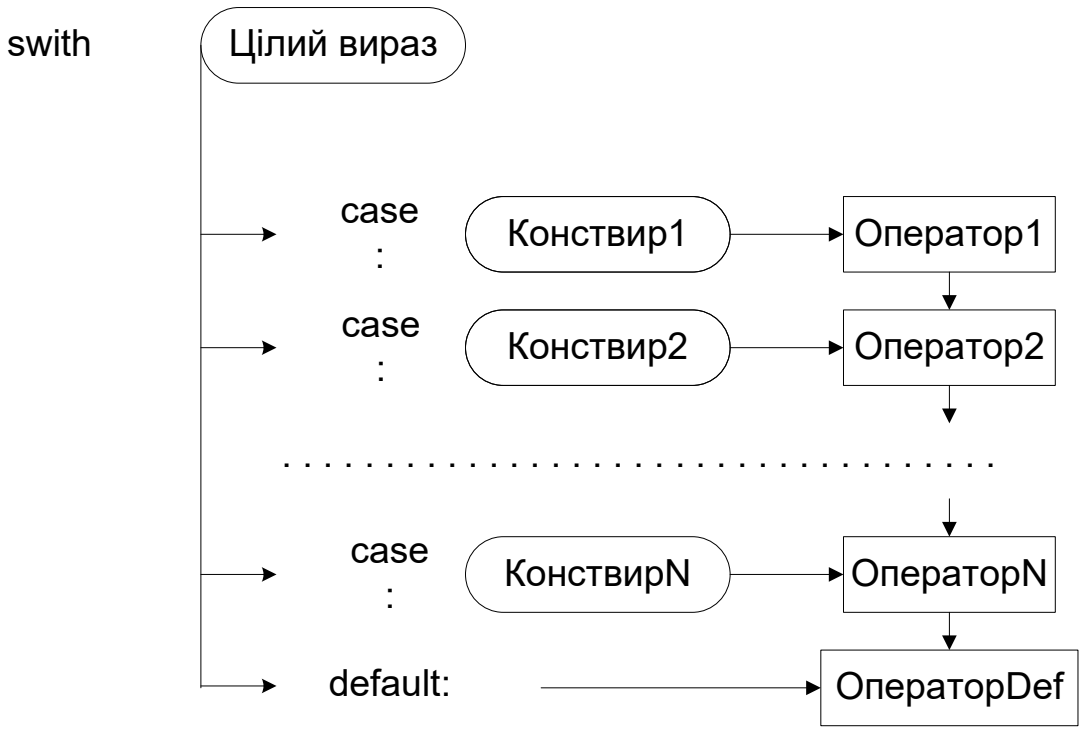


Рисунок 13.1 – Робота оператора варіанта

Ось простий приклад, у якому змінна `DayOfWeek` зберігає день тижня у вигляді цілого числа від 1 до 7, а нам треба одержати повідомлення про те, робочий це день або вихідний.

```
switch (DayOfWeek){
case 1: case 2: case 3: case 4:case 5:
printf("Робочий день\n"); break;
case 6: case 7:
printf("Вихідний день\n"); break;
default:
printf("День тижня заданий неправильно\n");
}
```

Увага!
Не забувайте завершувати варіанти оператором `break`!

14 ОПЕРАТОР ПЕРЕХОДУ `goto`

Оператор переходу `goto` виглядає дуже просто:

```
goto мітка;
```

Він викликає негайний перехід до деякого оператора, на який указує мітка. Мітка записується, як всі ідентифікатори, буквами, цифрами й знаками підкреслення, але не вимагає ніякого опису. Мітка ставиться перед оператором або відкриттям фігурних дужок й відділяється від них двокрапкою. Так утворюється позначений оператор або позначений блок. Ось класичний приклад зациклення, у якому змінна `x` буде нескінченно одержувати значення `0`:

```
M:  X = 0; goto M;
```

Можна написати навіть так:

```
M: goto M;
```

Ніякого порушення синтаксису тут немає, компілятор нічого не повідомить, він не вважає це помилкою. Раз людина так написала, отже, треба згенерувати машинний код і виконати його.

Подібні помилки зустрічаються напрочуд часто, звичайно, не в такому простому вигляді, як написано вище. Ніхто у тверезому розумі й доброму здоров'ї не напише такий код, але якщо оператор `goto` і позначений оператор знаходяться на достатньому видаленні один від одного, рядків на 50-70, то подібну помилку допустити дуже легко, а знайти й виправити дуже важко.

Крім того, різкий перехід до довільного оператора порушує плавний перебіг програми, ускладнює спостереження за її виконанням і сприяє появі помилок. Все це робить оператор `goto` небажаним для використання.

У 70-х роках минулого століття відбувалася бурхлива дискусія із приводу оператора `goto`. Супротивники ратували за повне вигнання його з мов програмування, вони придумали для його заміни оператори `continue` і `break`, доводячи, що їх цілком достатньо для зміни ходу програми. Вони оголосили, що кваліфікація програміста пропорційно обернена кількості операторів `goto` у написаному ним коді. Захисники наводили приклади ситуацій, у яких він необхідний. Дискусія закінчилася компромісом. Було вирішено, що оператор `goto` у деяких ситуаціях корисний, але краще ним не зловживати.

Вправи

1 Знайдіть алгоритм, що підраховує кількість цифр заданого цілого числа.

2 Знайдіть алгоритм, що дає суму цифр заданого цілого числа.

3 Ціле шестизначне число вважається "щасливим", якщо сума трьох його старших цифр дорівнює сумі трьох молодших цифр. Створіть алгоритм, що визначає, чи буде задане ціле число "щасливим".

Контрольні питання

1 Як можна записати розгалуження на три гілки за допомогою умовного оператора `if`?

2 Дано три змінні `a, b, c`. Як найменшим числом умовних операторів поміняти їхні значення так, щоб у результаті `a<b<c`?

3 Чи можна в мові С записати оператор циклу `for ( ; ;) x = 0;`?

4 Скільки разів буде виконаний цикл `while(k++<10) k++;` якщо перед циклом `k=1`?

5 Яким способом можна припинити виконання циклу?

6 Наведіть приклади зациклення.

7 Чи обов'язкова конструкція `default: оператор;` в операторі варіанта `switch`?

8 Що трапиться, якщо поставити конструкцію `default: оператор;` першим варіантом оператора `switch`?

9 Яку роль відіграють оператори `break` в операторі варіанта `switch`?

15 МАСИВИ Й ПОКАЖЧИКИ

Багато задач, рішення яких знаходиться за допомогою комп'ютера, вимагають перегляду й обробки великої кількості однотипних даних. Такі дані в науці й техніці позначаються змінними з індексами: x_1, a_0, x_k, M_y і т. д. Для прискорення перегляду однотипні дані найкраще помістити в сусідні комірки

оперативної пам'яті, щоб не витрачати час на їхній пошук. Так виникає поняття масиву.

Масив – це сукупність змінних одного типу, що зберігаються в суміжних комірках оперативної пам'яті.

Це визначення призводить до таких особливостей масиву:

- у всіх змінних одного типу, що утворюють масив, однакова довжина. Для переходу до наступного значення досить пересунутися по оперативній пам'яті на фіксовану кількість байтів, що дорівнює довжині типу. Це дуже швидка операція;

- оскільки тип всіх елементів масиву однаковий, вони позначаються одним і тим самим ім'ям, а розрізняються за допомогою індексів;

- для зберігання масиву треба виділити безперервну ділянку оперативної пам'яті. Тому масив повинен бути оголошений заздалегідь; при оголошенні треба вказати тип елементів масиву і їхню кількість, щоб можна було підрахувати розмір виділюваної ділянки оперативної пам'яті.

Отже, кількість елементів масиву, називана його довжиною, повинна бути відома заздалегідь. Це досить сильне обмеження. У більшості задач кількість елементів, які треба обробити за допомогою масиву, невідома заздалегідь. У таких випадках доводиться створювати масив з найбільшою передбачуваною довжиною, що призводить до перевитрати оперативної пам'яті. Деякі мови програмування дозволяють визначати масиви зі змінною довжиною, але й у їхній основі завжди лежить виділення максимальної кількості оперативної пам'яті.

15.1 Масиви

У мові С масиви створюються звичайними операторами визначення типу разом зі звичайними змінними або в окремих операторах. При визначенні масиву після його імені у квадратних скобках записується його довжина. Наприклад,

```
double a[5], b[128], c1 = 0.0;
char c[1000];
int p[100], q[100];
double x = 1.0, y[1024];
```

Індекси масиву завжди починаються з 0, збільшуючись на 1, так що останнє значення індексу менше його довжини на одиницю. Так, змінна з найбільшим індексом у визначених вище масивах – це `a[4]`, `b[127]` `c[999]`, `p[99]`, `q[99]`, `y[1023]`. Спочатку це спантеличує. Визначений вище масив a складається з п'яти змінних `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[4]`. Елемента `a[5]` у масиві немає!

Справа погіршується тим, що компілятор мови С не перевіряє границі масиву. Ви можете використовувати в програмі змінну `a[5]`. Компілятор не повідомить про помилку. Просто для `a[5]` буде взяте значення, що зберігається в наступній після `a[4]` комірці. Як правило, там лежить якесь «сміття», яке і виявиться значенням змінної `a[5]`. Добре, якщо це значення різко відрізняється від всіх інших значень масиву, і ви помітили помилку. Гірше, якщо воно порівнянно з іншими значеннями. Програма може довго працювати й давати неправильні результати, перш ніж ви побачите свою помилку.

Індекси можна задавати будь-якими цілочисельними виразами, наприклад `a[i+j]`, `a[i%5]`, `a[++i]`, `b['?']`, `x['9'-'0']`. При цьому треба ще ретельніше стежити за тим, щоб індекс не виявився негативним і не перевершив оголошену границю масиву.

Перед використанням масиву треба дати всім його елементам початкові значення, наприклад:

```
    a[0] = 0.01; a[1] = -3.4; a[2] = 2.89;
a[3] = 4.5; a[4] = -6.7;
    for (int i = 0; i < 128; i++) b[i] = 1.0 /
i;
    for (int i=0;i<100;i++) p[i]=2*i+1;
```

Якщо цього не зробити, то в елементах масиву виявиться сміття, узяте з відповідних комірок оперативної пам'яті. У деяких випадках елементи масиву автоматично обнуляються, але про це ми поговоримо пізніше.

При оголошенні масиву можна відразу задати початкові значення його елементів, записавши їх у фігурних дужках через кому у вигляді констант або константних виразів. При цьому

навіть необов'язково вказувати кількість елементів масиву, його довжина буде дорівнювати кількості початкових значень:

```
double a[]={0.01,-0.02,0.89,44.5e-3,-68.75};
```

Тут визначений масив a з 5 елементів – речовинних змінних подвоєної точності. Їхні початкові значення: a[0]==0.01; a[1]== -0.02; a[2]== 0.89; a[3] = = 44.5e-3; a[4] == -68.75;

Якщо ж довжина масиву указана явно, то кількість початкових значень не повинна її перевищувати. Можна задати менше початкових значень, ніж довжина масиву. Ці початкові значення привласнять перші елементи масиву, а останні елементи одержать нульове значення. Ось, наприклад, масив з 10 символьних змінних:

```
char c[10] = {‘a’, ‘b’, ‘c’};
```

Тут задані значення c[0] == 'a', c[1] == 'b' і c[2] == ‘c’. Інші елементи c[3], c[4],..., c[9] дорівнюватимуть ‘\0’.

Елементи масиву – це звичайні змінні свого типу й з ними можна виконувати всі операції, що входять у цей тип. Наприклад:

```
a[1] += 2*a[k]-3*a[k+1]/a[k-1];
```

Обробка масиву, як правило, виконується в циклах. Ось як звичайно обчислюється сума всіх елементів масиву:

```
double s=0.0;
int k;
for (k= 0 ;k < 128; k++) s += b[k];
```

У лістингу 7 показано, як можна обчислити діапазон значень масиву.

Лістинг 7. Обчислення діапазону значень масиву

```
#include <stdio.h>
#include <math.h>
#define LENGTH 100
main() {
double a [LENGTH], x = 0.1234;
```

```
int i;
for (i=0;i<LENGTH;i++)a[i]=i*sin(M_PI*i*x)
double amin = a[0], amax = amin;
for (i=1;i<LENGTH;i++){
if (a[i]<amin) amin=a[i];
if (a[i]>amax) amax=a[i];
}
double range = amax - amin;
printf("Діапазон значень = %f\n", range);}
```

Тут використана константа `M_PI`, визначена у файлі `math.h` і що дає наближене значення числа π.

Дуже часто треба знати не значення найбільшого або найменшого елемента масиву, а тільки його індекс. Ось як можна його знайти:

```
int i, ind = 0;
for (i = 1; i < 5; i++)
if (a[i] > a[ind]) ind = i;
```

Дуже часто доводиться виводити на консоль всі значення якого-небудь масиву. При цьому зручно вивести на одному рядку кілька значень Наступний приклад показує, як можна вивести масив `a` по п'ять значень у рядку:

```
for (k = 0; k < LENGTH; k++){
 printf ("a [%d] = %f ", k, a[k]);
 if (k % 5 == 4) printf("\n");
}
```

Вправи

1 Знайдіть суму абсолютних значень всіх елементів масиву.

2 Знайдіть найбільший і найменший за абсолютною величиною елемент масиву, а також їхні індекси.

3 Підрахуйте кількість елементів масиву, що перевищують за абсолютною величиною найбільший з перших десяти елементів масиву.

4 Знайдіть у масиві задане значення x. Повідомте його індекс або -1, якщо такого значення в масиві немає.

15.2 Багатомірні масиви

Елементами масивів можуть бути не тільки змінні простих типів, але й знову масиви. Можна оголосити:

```
char c[3][4];
```

Компілятор зрозуміє ці записи як визначення масиву `c` із трьох елементів, причому кожний елемент `c[0], c[1]` і `c[2]` – це масив типу `char`, що складається із чотирьох елементів. Зокрема `c[0]` – це ім'я масиву, елементи якого, як звичайно, розрізняються індексами, записаними у квадратних дужках: `c[0][0], c[0][i], c[0][2], c[0][3].`

Кількість індексів масиву називається його розмірністю. При визначенні багатомірного масиву можна задати початкові значення його елементам за тим самим принципом, що й в одномірного масиву:

```
int inds [][] = {{1, 2, 3}, {4, 5, 6}};
```

Визначений тут двовимірний масив `inds` складається із двох масивів по три елементи кожний. Після цього визначення елемент `inds [0][0]` буде дорівнювати 1, елемент `inds[0] [l]` дорівнює 2 і т. д. Елемент `inds[l][l]` буде дорівнювати 5.

Як бачите, двовимірний масив повністю відповідає математичному поняттю прямокутної матриці, тільки індекси матриці звичайно починаються з 1, а індекси двовимірного масиву починаються з нуля.

З визначення багатомірного масиву випливає, що його елементи розташовуються в оперативній пам'яті рядками. Елементи одного рядка зберігаються в сусідніх комірках, але елементи одного стовпця можуть перебувати далеко один від одного. Тому багатомірні масиви швидше проглядаються по рядках, а не по стовпцях. Для перегляду багатомірного масиву доводиться записувати вкладені цикли. Швидкий перегляд сусідніх комірок буде забезпечений, якщо перший індекс буде змінюватися в зовнішньому циклі, а останній індекс – у внутрішньому циклі. Ось як можна вивести значення двовимірного масиву по рядках у вигляді таблиці чисел:

```
for (i = 0; i < FIRST_DIM; i++){
```

```
 for (j = 0; j < SECOND_DIM; j++)
 printf ("%f ", a[i][j]);
 printf ("\n");
}
```

У лістингу 8 наведено приклад програми, що обчислює перші LINES рядків трикутника Паскаля, заносить їх у двовимірний масив і виводить його елементи на екран.

Лістинг 8. Трикутник Паскаля

```
#include <stdio.h>
#define LINES 10
main () {
 int p[LINES][LINES],i,j;
 for (i = 0; i < LINES; i++)
   for (j = 0; j < LINES; j++)
    p[i][j] = 0; // Обнуляємо масив.
printf("%3d\n", p[0][0] = 1);
p[1][0] = p[1][1] '= 1;
printf("%3d %3d\n", p[1][0], p[1][1]);
for (i =2; i < LINES; i++){
  printf("%3d ", p[i][0] = 1);
 for (j = 1; j < i; j++)
printf("%3d  ",p[i][j]=p[i-1][j-1]+p[i-1][j]);
printf("%3d\n",p[i][i]=1);
  }
 }
```

Вправи

5 Якщо ви знайомі з лінійною алгеброю, то напишіть програму множення двох матриць.

6 Якщо ви добре знайомі з лінійною алгеброю, то напишіть програму обертання квадратної матриці.

7 Якщо ви любите лінійну алгебру, то опишіть мовою С всі дії з матрицями.

Так само можна визначити тривимірний масив:

```
int a[2] [3] [2] =
```

```
{{{1,2},(3,4}},{{{1,2},{3,4}},{{{1,2},{3,4}},
{{1,2},{3,4}},{{{1,2},{3,4}},{{{1,2),{3,4}}};
```

Припустимо й більше число індексів. Стандарт мови С не накладає обмеження на розмірність масиву, але кожний компілятор задає максимальну кількість індексів масиву. Вона звичайно вимірюється десятками й ніколи не досягається в реальних програмах. На практиці рідко зустрічаються масиви більш ніж із трьома індексами.

Якщо при визначенні багатомірного масиву задані початкові значення, то кількість елементів за першим індексом можна не вказувати. Компілятор зможе відновити його за наступними розмірностями. Наприклад, попереднє визначення можна записати так:

```
  int a [][3][2] =
{{{1,2},{3,4}},{{{1,2},{3,4}},{{{1,2},{3,4}},
{{1,2},{3,4}},{{{1,2},{3,4}},{{{1,2},{3,4}}};
```

15.3 Покажчики

Дотепер ми мали справу зі змінними простих типів і масивами, складеними з таких змінних. У мові С дуже часто застосовуються змінні, значення яких – це адреси комірок оперативної пам'яті. Такі змінні називаються покажчиками (pointers).

Нагадаємо, що коміркою оперативної пам'яті ми називаємо сукупність суміжних байтів, що містять одиницю інформації: число, символ, рядок.

Довжина комірки, тобто кількість байтів, що складають комірку, може бути різною: один, два, чотири, вісім байтів залежно від типу інформації, що зберігається в комірці. Адреса комірки – це порядковий номер першого з її байтів.

Найчастіше в комірці зберігається значення якого-небудь типу, тому в мові С говорять не просто "покажчик", а "покажчик на такий-то тип". Це позначається на визначенні, що виглядає так:

```
char  *pc;
doule  *pd;
```

Зірочка позначає покажчик, щоб відрізнити його від звичайної змінної. Адже в одному визначенні можуть зустрітися й покажчики, і звичайні змінні:

```
char *pc1, c0 = '?', *pc2;
int *pi, k = 5;
```

У деяких випадках заздалегідь неможливо задати тип для покажчика. У таких випадках робиться запис:

```
void *p;
```

Такий покажчик не знає структуру й розмір комірок пам'яті. Після визначення його значення він просто буде містити номер деякого байта. Перед його використанням треба зробити явне приведення до визначеного типу.

Наведені вище оператори визначають тільки покажчики, наприклад:

```
int *pi;
```

У цьому випадку компілятор виділяє пам'ять для покажчика `pi`. У ній буде зберігатися адреса комірки, на яку буде посилатися покажчик, але ніякої адреси поки ще немає – покажчик ні на що не вказує. Яким-небудь чином йому треба дати значення – адресу якоїсь комірки. Можна записати адресу явно: `pi = (int *)245040;` але робити так ніколи не доводиться, адже оперативна пам'ять розподіляється операційною системою, ми не знаємо, яка адреса у потрібної нам комірки.

Набагато частіше покажчику дається адреса комірки пам'яті, зайнятої якоюсь уже визначеною до цього змінною простого типу. Для цього треба знати цю адресу, а її можна одержати операцією *узяття адреси* змінної. Ця операція позначається амперсандом, записуваним перед змінною, можемо написати присвоювання покажчику адреси змінної k:

```
pi = &k;
```

Тепер покажчик `pi` повністю визначений, він показує на комірку пам'яті, зайняту змінною `k`, містить її адресу. Це схематично показано на рисунку 15.1, де для прикладу взята конкретна адреса 200.

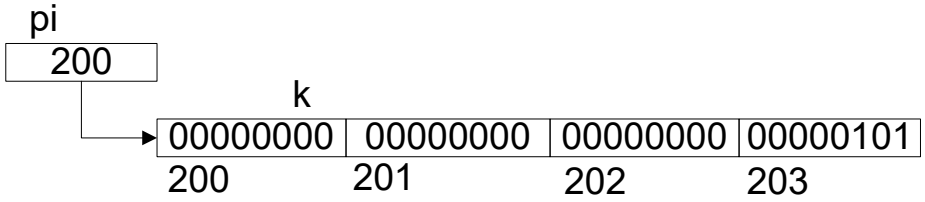


Рисунок 15.1 – Визначення покажчика

Ми одержали те, що в програмуванні називається непрямою адресацією. За покажчиком визначається не значення змінної, а тільки її адреса, за якою перебуває змінна. До такого подвійного визначення потрібно звикнути. Спочатку буває важко розібратися в хитросплетеннях покажчиків, тим більше, що багато програмістів грішать цим.

Ми можемо витягти вміст комірки, адреса якої міститься в покажчику, скориставшись операцією узяття змінної за адресою. Для цієї операції не придумано спеціального позначення, перед покажчиком знов-таки записується зірочка.

```
int n = *pi;
```

Після цього змінна `n` дорівнюватиме 5. Операція "зірочка" створює повноцінну змінну. Ви можете вважати, що ім'я змінної просто починається із зірочки, хоча насправді це результат операції, а не ім'я. Такий запис можна використовувати у виразах: `*pi / 2 + k`. Можна зробити присвоювання: `*pi = 10` і всі інші операції, доступні цій змінній.

От тому-то зірочка записується у визначенні покажчика. Рядок визначення `int *pi;` варто читати так: "`int` – це тип результату обчислення виразу `*pi`".

Отже, у мові С є операція узяття адреси і є операція узяття вмісту комірки за адресою. Ці дві операції взаємно зворотні, як додавання й віднімання. Вони виконуються зправа наліво і їх можна застосувати разом. Ми можемо написати:

```
char ch = *&c0;
```

При цьому ми одержимо адресу, а потім значення символьної змінної, або написати:

```
pc2 = &*pc1;
```

У цьому випадку ми одержимо спочатку значення за адресою, а потім знову адресу. Треба підкреслити, що ці операції застосовуються тільки до змінних, але не до констант або виразів, не можна написати `&(a+b)`, тому що результат виразу заноситься в тимчасову комірку, що не має постійної адреси.

При роботі з покажчиками варто уважно стежити за тим, щоб у них було визначене значення. Після початкового визначення – `int *pi;` – покажчик містить сміття, що залишилося у виділеній для нього комірці після попередніх обчислень. Добре, якщо цей вміст не є ніякою адресою оперативної пам'яті. Набагато гірше, якщо він випадково виявиться адресою комірки, важливої для роботи комп'ютера. Тоді будь-яким присвоюванням типу `*pi=x;` ми змінимо вміст цієї комірки й порушимо роботу програми. Щоб уникнути такої ситуації, називаної "висячим покажчиком", у мову С введено поняття нульового покажчика, значення якого – така адреса, якої немає в оперативній пам'яті. Це значення записується константою `NULL`. Визначаючи покажчик, якому ми не можемо відразу ж дати точне значення, варто завжди давати йому значення `NULL`:

```
int *pi = NULL;
```

Замість константи `NULL` можна записати просто `0`:

```
int *pi = 0;
```

Не думайте, що це буде адреса нульового байта. Компілятор перетворить такий запис за своїми правилами й може занести в комірку, виділену для покажчика `pi` інше значення, що не відповідає жодній комірці оперативної пам'яті.

16 АДРЕСНА АРИФМЕТИКА

З викладеного вище видно, яку свободу дає мова С програмісту в роботі з оперативною пам'яттю. Програміст може звернутися до будь-якої комірки й спробувати змінити її значення. Насправді така спроба частіше за все виявиться невдалою. Захистом оперативної пам'яті займається операційна система. Вона стежить за тим, щоб програма могла змінити

тільки виділену для неї пам'ять і не змінювала вміст важливих для роботи комп'ютера ділянок. Але в оперативній пам'яті є загальні області, поділювані декількома програмами. Їхню роботу нерозумне звернення до оперативної пам'яті може порушити.

Крім того, мова С пропонує набір арифметичних операцій з покажчиками, що дозволяють переміщатися по оперативній пам'яті й полегшують безпосередню роботу з її комірками:

- до покажчика можна додати ціле значення, додатне або від'ємне. Додавання до покажчика 1, тобто операція `pi+1`, або збільшення покажчика на 1, тобто операції `pi++`, `++pi`, призводять до переходу покажчика на наступне значення свого типу. Важливо зрозуміти, що збільшення адреси на 1 означає не збільшення номера байта на 1, а додаток до поточного номера байта довжини типу покажчика. У прикладі рисунка 15.1 значення `pi+1` буде більше значення `pi` на 4 байти й стане показувати на байт із номером 204. Збільшення `pi` на 2, тобто `pi+2`, дасть адресу 208 і т. д. Збільшення або зменшення рівно на один байт буде зроблено тільки для типу `(char*)`. Операція додавання до покажчика цілого значення комутативна, можна записати як `pi+k`, так і `k+pi`;

- з покажчика з більшою адресою можна відняти покажчик того самого типу з меншою адресою. Результатом буде число значень даного типу, що лежать між цими адресами. Наприклад, якщо `p1` і `p2` - покажчики на цілий тип `int`, що має довжину 4 байти, покажчик `p1` містить значення 200, а покажчик `p2` – значення 208, то різниця `p2-p1` буде дорівнювати 2;

- як уже зазначено раніше, до покажчика можна застосувати операції інкремента й декремента, префіксні й постфиксні: `pi++`, `++pi`, `pi--`, `--pi`. Вони дають у результаті покажчик на наступне або попереднє значення свого типу;

- з покажчиками можна робити присвоювання виду `=`, `+=` або `-=`, якщо у правій частині присвоювання знаходиться вираз того самого типу, що й покажчик у лівій частині;

- вирази, що складаються з покажчиків того самого типу, можна порівнювати між собою операціями `==`, `!=`, `<`, `<=`, `>`, `>=`. Покажчики вважаються рівними, якщо містять одну й ту саму адресу, показують на одну й ту саму комірку пам'яті.

Більший покажчик той, котрий містить більшу адресу. Будь-який покажчик можна зрівняти з нульовим покажчиком на рівність або нерівність;

- до покажчика можна застосувати операцію заперечення `!pi`. Вона переводить ненульовий покажчик у нульовий і назад.

Треба попередити, що адресна арифметика – це джерело безлічі помилок, зроблених у свій час у тисячах програм. Записуючи операції з покажчиками, треба повністю усвідомлювати про їхні результати. Зокрема вирахування й порівняння покажчиків варто виконувати тільки в межах одного масиву.

Участь покажчиків у виразах порушує питання про пріоритет операцій узяття адреси `&` і узяття змінної за адресою `*`. Це операції із досить високим пріоритетом. Вони виконуються після інкременту й декременту, але до явного перетворення типу й множення/ділення. Як уже було сказано, вони виконуються справа наліво. Тут треба бути уважним. У програмах, написаних мовою С, ви часто побачите такий запис:

```
*pi++;
```

Тут береться значення за адресою, що зберігається в покажчику `pi`, а потім значення покажчика збільшується. Чому, адже інкремент виконується раніше від узяття значення за адресою? Розглянемо це докладніше. Нехай записані оператори:

```
int a=5, *p=&a;
int b=*p++;
```

Після цього значення змінної `b` дорівнює 5. Порядок операцій у другому операторі такий: `*(p++)`, але інкремент постфіксний, тому збільшення адреси відбувається після обчислення всього виразу.

16.1 Зв'язок масивів і покажчиків

У мові С ім'я масиву завжди є покажчиком на початок масиву, адресою його першого елемента, наприклад:

```
int a [10];
```

При такому визначенні компілятор неявно визначає покажчик з іменем, що збігається з іменем масиву, і зберігає в ньому адресу масиву, як показано на рисунку 16.1, де для прикладу взяте конкретне значення адреси 200.

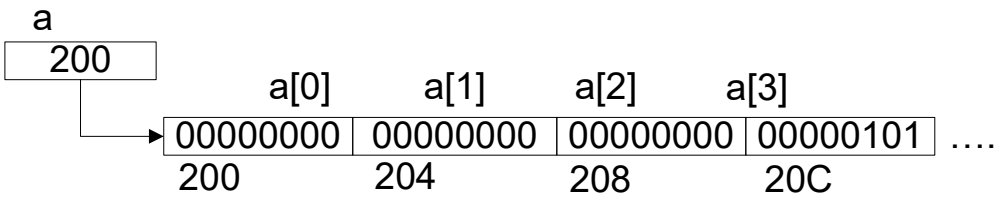


Рисунок 16.1 – Визначення масиву `int a[10]`

Насправді конкретні адреси визначає операційна система.

Іменем масиву можна вільно користуватися як покажчиком. Замість a[0] можна написати `*a`, замість `a[1] – *(a+1)`, замість `a[k] – *(a+k)`. Такий запис можна робити у виразах:

```
*(a+3)+=*a-*(a+1)/*(a+2);
```

Більш того, компілятор так і робить, він одразу ж перетворює запис із індексами в запис із покажчиками й потім працює із цим записом. Тепер можна зрозуміти, чому в мові С індекси елементів масиву завжди починаються з нуля. Індекс – це просто зміщення по оперативній пам'яті від початку масиву. Оскільки додавання цілого значення до покажчика – комутативна операція, справедливий кожний з таких записів:

```
a[2]==*(a+2)==*(2+a)==2[a].
```

Перевірте, кожний із цих виразів припустимий!

Є одне істотне розходження між іменем масиву й покажчиком, визначеним зірочкою: ім'я масиву – це покажчик-константа. Масив розміщається в пам'яті операційною системою, вона визначає його адресу, і цю адресу не можна змінити. Не можна написати, наприклад, `a=&x`. За необхідності роботи з покажчиком-змінною можна визначити її й зв'язати з масивом:

```
int *p = a;
```

Після такого визначення покажчик `p` показує на ту саму комірку пам'яті, що й `a`, отже, `*p==a[0], *(p+1)==a[1]` і т. д. Цікаво те, що можна записувати `p[0], p[1],` хоча ми не визначали ніякого масиву `p`! Це припустимо, тому що компілятор механічно перетворить індексний запис у запис із покажчиками.

Далі в програмі можна зробити нове визначення:

```
int b[1000];
```

Якщо перекинути покажчик на новий масив `p=b,` то `*p==p[0]==b[0].`

Покажчики дуже часто застосовуються для роботи з масивами. От як можна обчислити суму елементів масиву за допомогою покажчиків:

```
double a[100], s=0.0, *p=NULL;
//Якимось чином визначаємо елементи масиву a
int k=0;
for (p=a; k<100; p++, k++) s+=*p;
```

Дотепер ми визначали покажчик на масив. Якщо буде потреба, можна визначити масив покажчиків, наприклад:

```
int k=22;
int *p[5]={&k, &k, &k, &k, &k};
```

Тут визначений масив з п'яти покажчиків. Всі покажчики містять адресу змінної k. Ще більш цікава картина виникає при визначенні багатомірного масиву:

```
int a[2][3];
```

При цьому виникає схема, показана на рисунку 16.2.

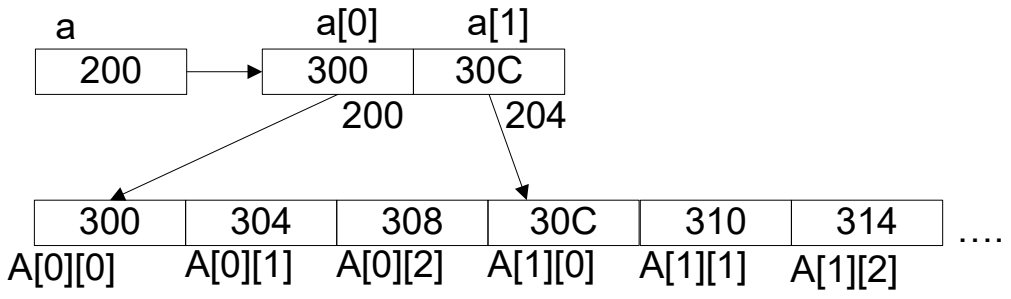


Рисунок 16.2 – Визначення двовимірного масиву

Тут ім'я масиву a – це покажчик-константа на масив із двох елементів a[0] і a[1]. Це масив покажчиків-констант, кожний його елемент – покажчик на масив із трьох елементів. Всіма покажчиками можна користуватися; більш того, тут виникає нове поняття – покажчик на покажчик. Справді, покажчик a показує не на змінну звичайного простого типу, а на покажчик a[0]. Отже, *a – це a[0], але *a[0] – це a[0][0]. Звідси **a – це a[0][0]. Отже,

a – покажчик на початок масиву покажчиків a[0];

*a==a[0] – покажчик на елемент a[0][0];

**a – сам елемент a[0][0].

Далі

a+1 – покажчик на елемент a[1];

*(a+1) – сам елемент a[1] – покажчик на елемент a[1][0];

*(a+1)+2 – покажчик на елемент a[1][2];

((a+1)+2) – сам елемент a[1][2].

Отже, елемент масиву a[i][j] компілятор запише як **(*(a+i)+j). У лістингу 9 показано, як всі ці адреси можна вивести на екран дисплея.

Лістинг 9. Адреси оперативної пам'яті

```
include <stdio.h>
main(){
int a[2][3]={{1, 2, 3}, {4, 5, 6}};
printf("Адреса масиву a: %d\n", a);
printf("Адреса елемента a+1: %d\n", a+1);
printf("Адреса елемента *(a+1):%d\n",*(a+1));
printf("Адреса елемента *(a+1)+2: %d\n",
*(a+1)+2);
printf("Елемент *(*(a+1)+2): %d\n",
*(*(a+1)+2));
}
```

Змінну типу "покажчик на покажчик на якийсь тип" можна визначити і явно:

```
int **p;
```

Ми одержуємо ***подвійну непряму адресацію***. Покажчик `p` містить адресу не простої змінної, а інший покажчик, що містить адресу змінної. Потім можна дати покажчику `p` значення вже визначеного подвійного покажчика, наприклад, `p=a,` і працювати з матрицею `a` за допомогою покажчика `p`.

Отже, у мові С можна працювати з подвійними, потрійними й більше покажчиками, реалізуючи множинну непряму адресацію, можна в будь-якому сполученні визначати масиви покажчиків і покажчики на масиви, виконуючи складну обробку комірок оперативної пам'яті. Не захоплюйтеся цими можливостями, щоб не ускладнювати програму.

16.3 Робота з рядками

Ми вже зустрічалися зі рядковими константами виду `"Рядок символів"`. Компілятор розглядає рядок як масив символів, завершений нуль-символом. Цей запис визначає простий масив символів:

```
char c[] = {'Р','я','д','о','к',' ',
'с','и','м','в','о','л','і','в'};
```

А ця визначає рядок:

```
char s[] ={'Р,'я','д','о','к',' ',
'с','и','м','в','о','л','і','в','\0'};
```

Компілятор завжди генерує на місці рядкової константи її адресу. Цю адресу може присвоїти покажчик, отже, можна записати:

```
char *p="Рядок символів";
```

Визначення `s` створило покажчик-константу на рядок, а визначення `p` – покажчик-змінну, котрій дана адреса рядкової константи. В іншому ці покажчики рівноправні. Ми могли б визначити рядок `s` і так:

```
char s[]="Рядок символів";
```

Немає ніякої різниці між двома визначеннями рядка s, але між визначеннями p і s є та сама різниця, що була при визначенні масивів: p – це покажчик-змінна, s – покажчик-константа.

Стандарт мови С не дозволяє змінювати рядкову константу через покажчик на неї. На практиці компілятор звичайно розташовує константу в області пам'яті, доступній вашій програмі тільки для читання, але не для запису. За допомогою покажчика p ви не зможете змінити рядок. Для рядка s, визначеного як масив з нуль-символом наприкінці, такий компілятор виділить пам'ять в області вашої програми й ви вільні розпоряджатися рядком на власний розсуд.

Проте покажчики дуже широко застосовуються для роботи з рядками. Ось як можна підрахувати кількість символів у рядку – її довжину:

```
int len=0;
char *p="Рядок символів, довжину якого ми підраховуємо";
for ( ;*p;p++) len++;
```

Цикл можна записати іншим оператором:

```
while (*p) {p++; len++;}
```

Або навіть так:

```
while (*p++) len++;
```

Такий же лаконічний запис можна зробити в операторі циклу for:

```
for (  ; *p++;) len++;
```

Можна скористатися вирахуванням покажчиків і підрахувати довжину рядка так:

```
int len=0;
char *p="Рядок символів, довжину якого ми підраховуємо";
char *q=p;
while (*q++);
len=q-p;
```

У всіх випадках завершальний нуль-символ не враховується при підрахунку, і як тільки *p стане рівним \0, виконання циклу припиниться, ми одержимо довжину len, що дорівнює 46.

У програмах мовою С ви побачите приклади дуже лаконічного запису операцій з покажчиками, більш схожі на шифровку. Ось класичний приклад копіювання всіх символів рядка, на який вказує q, у рядок p:

```
char p[1000];
char *q="Копійований рядок";
while (*p++=*q++);
```

Тут в одному виразі відбувається й переміщення покажчиків, і присвоєння символів, і перевірка закінчення рядка. З рядком можна працювати як з масивом, використовуючи індекси. Так, у визначеному вище рядку p[0]=='С', p[1]=='т', p[10]=='в'. Останній символ рядка м – це елемент p[45]. Завершальний нуль-символ – це елемент масиву p[46]. Ми можемо звернутися до будь-якого символу за індексом або змінити будь-який символ простим присвоюванням виду p[k]='?';

Як приклад роботи з рядками наведемо програму, що підраховує кількість поява кожного символу в заданому рядку.

Лістинг 10. Підрахунок кількості поява кожного символу в рядку

```
#include <stdio.h>
#include <string.h>
main(){
char s[]="Підрахуємо кількість появ кожного символу.";
int ch[256], i;
for (i=0; i<256; i++) ch[i]=0;
for (i=0; i<strlen(s); i++) ch[s[i]]++;
for (i=32; i<256; i++)
printf ("Символ %c з'явився %d раз\n", i, ch[i]);
}
```

Вправи

1 Напишіть програму, що підраховує кількість пробілів у заданому тексті.

2 Напишіть програму, що підраховує кількість появ заданого символу в заданому тексті.

3 Напишіть програму, що підраховує кількість слів у заданому тексті. Ви можете використати алгоритм, зображений на блок-схемі рисунка 16.3.

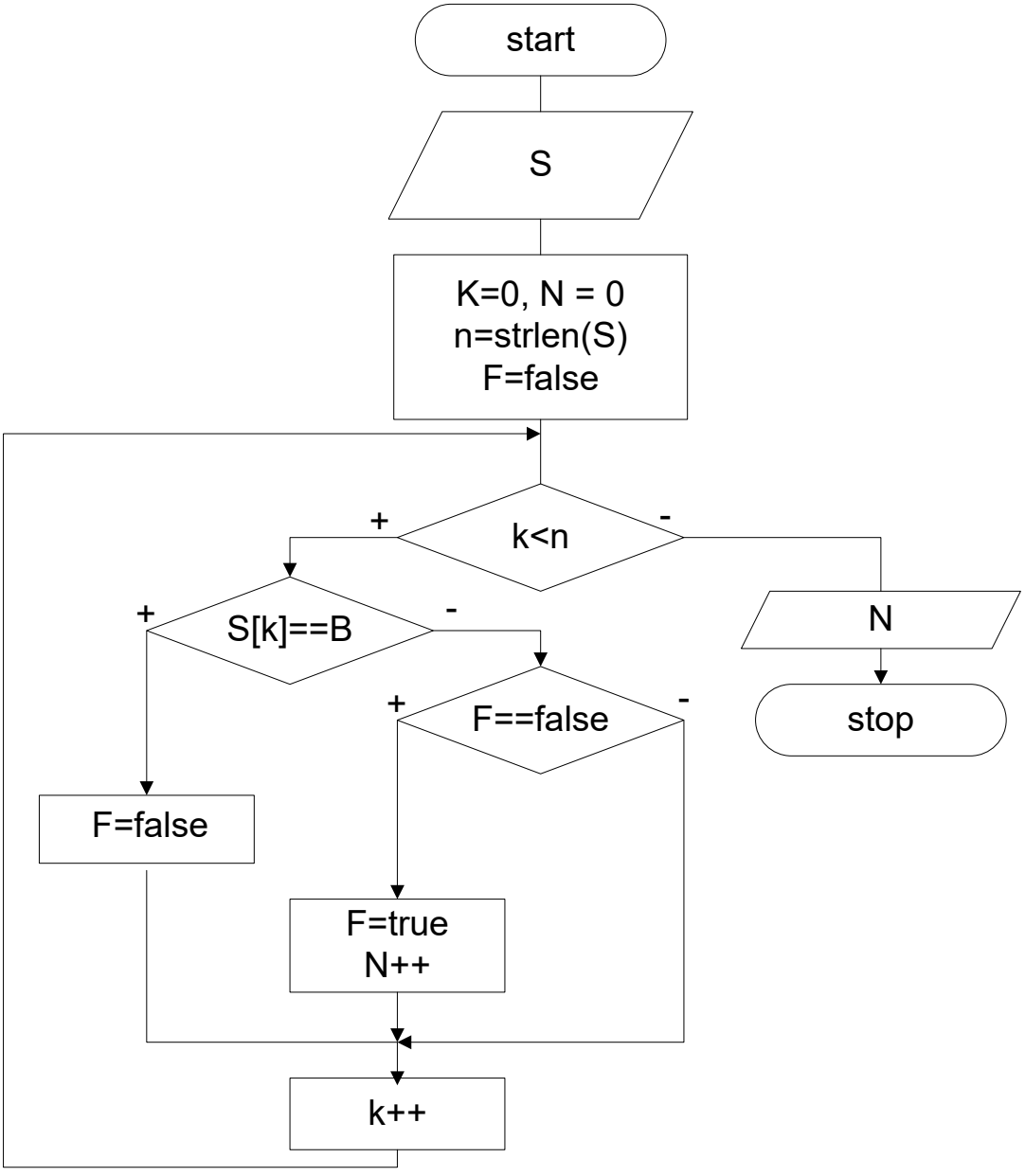


Рисунок 16.3 – Підрахунок кількості слів у тексті

4 Напишіть програму, що відшукує індекс заданого символу в заданому тексті. Якщо такого символу в тексті немає, то

програма повинна дати значення -1. Якщо символ зустрічається кілька разів, то можна написати два варіанти програми, що дають індекс першої або останньої появи символу в тексті.

5 Напишіть програму, що відшукує індекс, з якого починається задане слово в заданому тексті. Якщо такого слова в тексті немає, то програма повинна дати значення -1. Якщо слово зустрічається кілька разів, то можна написати два варіанти програми, що дають індекс першої або останньої появи слова в тексті.

17 ФУНКЦІЇ ОБРОБКИ РЯДКІВ

У стандарт мови С входить бібліотека функцій, призначених для роботи з рядками. У лістингу 10 ми вже використовували функцію `strlen()` із цієї бібліотеки. Опис цих функцій зроблений у файлі `string.h`, який треба включати в програму директивою `#include`, а самі функції, записані в машинному коді, зберігаються в бібліотеці `libc`. Бібліотека `libc` підключається до програми автоматично, ніяких додаткових вказівок компіляторові робити не треба.

Один з аргументів цих функцій – покажчик на вже визначений рядок символів. Функція переглядає рядок і дає відповідний результат. Дамо опис стандартних функцій.

Функція `strcat(si,s2)` доповнює перший рядок `s1` символами із другого рядка `s2`. Як результат, вона повертає адресу першого рядка, тобто значення покажчика `s1`. У першому рядку повинно бути досить місця для поміщення всіх символів другого рядка. Приклад:

```
char s[100];
strcat (s, "Початок рядка. ");
strcat (s, "Закінчення рядка.");
```

Функція `strncat(s1,s2,n)` доповнює перший рядок `s1` першими `n` символами із другого рядка `s2`. В іншому вона діє так само, як функція `strcat()`.

Функція strcmp(s1,s2) порівнює перший рядок s1 із другим рядком s2. Як результат, вона повертає 0, якщо рядки збігаються, або різницю кодів перших незбіжних символів. Приклад:

```
int r = strcmp("Іванов", "Іванцов");
```

Одержимо різницю кодів символів "о" і "ц". Для кодувань, у яких кирилиця розташована за алфавітом, одержимо значення r, що дорівнює -8.

Функція strncmp(s1,s2,n) порівнює перші n символів рядків s1 і s2. В іншому вона діє так само, як функція strcmp().

Функція strcpy(s1,s2) копіює другий рядок s2 у перший рядок s1. Старий вміст рядка s1 стирається. Як результат, функція повертає адресу першого рядка, тобто значення покажчика s1. У першому рядку має бути досить місця для поміщення всіх символів другого рядка. Приклад:

```
char s[100];
strcpy(s,"Цей рядок копіюється в рядок s.");
```

Функція strncpy(s1,s2,n) копіює перші n символів другого рядка s2 у перший рядок s1. В іншому вона діє так само, як функція strcpy().

Функція strlen(s) підраховує кількість символів рядка без завершального нуль-символу. Вона влаштована так, як було показано вище. Приклад:

```
int n=strlen("Підрахунок довжини цього рядка.");
```

Функція strchr(s,c) знаходить покажчик на місце першої появи символу c у рядку s. Якщо такого символу в рядку немає, функція дає NULL. Приклад:

```
char *p=NULL;
if((p=strchr("Іванов,Петров,Сидоров.",'П'))
!=NULL)
  printf("%s\n",p);
```

Функція strrchr(s,c) знаходить покажчик на місце останньої появи символу c у рядку s. В іншому вона діє так само, як функції strchr().

Функція strstr(s1,s2) знаходить покажчик на місце першої появи рядка s2 у рядку s1. Якщо такого підрядка в рядку s1 немає, функція повертає NULL. Приклад:

```
char *p=NULL;
if ((p=strstr("Іванов, Петров, Сидоров.",
"Петров"))!=NULL)
  printf("%s\n",p);
```

Функція strpbrk(si,s2) повертає покажчик на місце першої появи в першому рядку s1 кожного із символів другого рядка s2 або NULL, якщо в рядку s1 немає жодного символу з рядка s2. Приклад:

```
if (strpbrk("(x+2)*(y-1)","+-*/%")!=NULL)
  printf("Арифметичний вираз\n");
```

Функція strspn(s1, s2) підраховує довжину першого підрядка рядка s1 цілком, що складається із символів рядка, s2, узятих у будь-якому порядку. Приклад:

```
if (strspn("Рядок із пробілами","t\r\n")>1)
  printf("Зайві пробільні символи\n");
```

Функція strcspn(s1, s2) підраховує довжину першого підрядка рядка s1, що цілком складається із символів, відсутніх у рядку s2.

Функція strtok(s1, s2) повертає покажчик на перше слово рядка s1. Роздільники слів перераховані в рядку s2. Наступний виклик функції strtok (NULL,s2) дасть покажчик на друге слово того самого рядка, якщо її перший аргумент дорівнює NULL. Рядок роздільників s2 може змінюватися від виклику до виклику. Якщо в рядку вже немає слів, то вертається NULL. Приклад:

```
char *s="Перебираємо всі слова цього
рядка";
```

```
char *p = strtok(s, "\t\r\n");
printf("Перше слово: %s\n", p);
while ((p=Strtok(NULL, "\t\r\n"))!=NULL)
  printf("Наступні слова: %s\n", p);
```

17.1 Робота з окремими символами

Під час роботи з текстами дуже часто доводиться аналізувати окремі символи: відзначати переставляння рядків, виділяти речення, видаляти зайві пробіли. У стандартну поставку мови С входить бібліотека функцій, що здійснюють перевірки й найпростіші перетворення символів. Опис бібліотеки зроблений у файлі `ctype.h`, яку слід включити в програму директивою препроцесора `#inciude`.

У функції, що перевіряє символи, заголовок записується за принципом int isxxx(int c), де xxx замінюється коротким позначенням класу символів, приналежність до якого перевіряє функція. Функції повертають ненульове значення, якщо символ с належить зазначеному класу, і 0 в іншому випадку. Перелічимо ці функції.

– `isalpha(c)` – символ `c` – буква;

– `isdigit(c)` – символ `c` – арабська цифра від 0 до 9;

– `isalnum(c)` – символ `c` – буква або арабська цифра;

– `isxdigit(c)` – символ `c` може брати участь у записі шістнадцяткового числа, тобто це цифри від `0` до `9` і букви `A, B, C, D, E, F, a, b, c, d, e, f`;

– `isspace(c)` – символ `c` – пробільний, тобто пробіл, символ вертикальної або горизонтальної табуляції, символ переставлення рядка, повернення каретки, переведення сторінки;

– `ispunct(c)` – символ `c` – знак пунктуації;

– `isprint(c)` – символ `c` – друкований, що має графічне подання, включаючи пробіл;

– `isgraph(c)` – символ `c` – друкований, за винятком пробілу;

– `iscntrl (c)` – символ `c` – керуючий, тобто символ з кодом від 0 до 31 або символ з кодом 127;

– `isascii (c)` – символ `c` – символ ASCII, тобто символ

з кодом від 0 до 127;

– `isupper(c)` – символ `c` – велика буква;

– `islower(c)` – символ `c` – мала буква.

Ці функції застосовуються в програмах обробки тексту, у тому числі майже у всіх текстових редакторах. Спочатку вони були розраховані на англійські тексти. Наприклад, функція `isalpha()` вважала буквою тільки символи з кодами від 65 до 90 і від 97 до 122. Для російських текстів і текстів, написаних іншими мовами, така функція призведе до величезного числа помилок. Тому при локалізації С ці функції обов'язково переписуються з урахуванням особливостей мови. Для російської доводиться враховувати й різні кодування символів. У русифікованих програмах часто з'являються помилки через те, що вони застосовуються з не призначеною для них бібліотекою.

Друга група функцій виконує деякі перетворення символів.

– `toascii(c)` – перетворить ціле число с у символ ASCII, залишивши в ньому тільки 7 молодших бітів;

– `toiower(c)` – перетворить букву с у нижній регістр. Заголовні букви стають рядковими, інші символи не змінюються;

– `toupper(c)` – перетворить букву с у верхній регістр. Малі літери стають великими, інші символи не змінюються.

18 ДИНАМІЧНЕ ВИДІЛЕННЯ ПАМ'ЯТІ

Всі змінні й масиви, з якими ми працювали дотепер, визначалися операторами визначення типу. Ми описували їх перед компіляцією, до виконання програми. Для таких змінних оперативна пам'ять виділяється перед виконанням програми й зберігається до її закінчення. Це статичне виділення пам'яті.

У багатьох випадках заздалегідь неможливо вгадати, буде потрібно змінна чи ні, або визначити, скільки місця в оперативній пам'яті знадобиться для масиву. У таких випадках треба визначати змінну або масив динамічно, під час виконання програми, виділяти їй пам'ять безпосередньо перед її використанням і звільняти пам'ять, коли змінна вже не потрібна.

У мові С основний засіб динамічного виділення пам'яті – функція `malloc(size)` `(memory allocation)`.

Аргументом size цієї функції служить кількість байтів, яку необхідно виділити в оперативній пам'яті. Функція повертає покажчик типу (void*) на перший байт виділеної ділянки пам'яті або NULL, якщо виділити пам'ять не вдалося.

```
    char *p;
    if  ((p=(char*)malloc(1000))==NULL)  printf
("Недостача пам'яті\n");
    else strcpy(p, new_text);
```

Тут виділена пам'ять під символьний масив з 1000 елементів. Функція malloc() нічого не знає про потрібний тип покажчика й створює покажчик невизначеного типу (void*). Тому зроблено явне перетворення типу. Ще один приклад:

```
    double *p;
    if ((p=(double*)malloc(800)==NULL){
    printf("Недостача пам'яті\n");
    return;
    }
    int i;
    for (i=0; i<100; i++) p[i]=(double)i;
```

Тут пам'ять запитана з розрахунком, що довжина типу double дорівнює 8 байтів. Цей розрахунок не завжди правильний. Набагато краще узнати довжину типу, щоб точно вказати число байтів. Тут допомагає ще одна операція мови С – операція визначення довжини типу sizeof(тип). Вона повертає довжину заданого типу в байтах. З її використанням попередній приклад, нарешті, буде виглядати так, як він виглядає в більшості програм:

```
    double *p;
    if
  (p=(double*)malloc(100*sizeof(double)]==NULL)
    {
    printf("Недостача пам'яті\n");
    return;
    }
    int i;
```

```
for (i=0; i<100; i++) p[i]=(double)i;
```

При виклику функції `malloc()` треба враховувати, що вона завжди виділяє безперервну область пам'яті, відшукуючи підходящу за розміром ділянку серед вільних блоків пам'яті. Тому не варто без необхідності замовляти занадто багато пам'яті за один раз. Це прискорить пошук вільної ділянки й дозволить уникнути відмов у вигляді повернутого `NULL`.

Пам'ять, виділена функцією `malloc()`, залишається зайнятою до закінчення програми, хоча необхідність у ній може відпасти раніше. Краще звільняти її відразу ж після використання. Для звільнення пам'яті служить функція `free(pointer)`. Їй передається покажчик `pointer`, раніше визначений функцією `malloc()`. За цим покажчиком функція `free()` впізнає початок ділянки памяти, що звільняється, і його розмір.

Отже, динамічне виділення й звільнення ділянок оперативної пам'яті виконується парою функцій `malloc()`, `free()`. Викликавши першу функцію, треба відразу ж подумати про те, у який момент викликати другу, щоб вчасно й повністю звільнити пам'ять. Поширена помилка програмування, називана "втратою пам'яті" `(memory leak)`, полягає в тім, що пам'ять виділяється динамічно в циклі, але не звільняється. Поступово вільна пам'ять закінчується, хоча є багато невикористовуваних ділянок, і програма завершується аварійно.

19 БІНАРНИЙ ПОШУК У МАСИВІ

Задача пошуку інформації з якихось її ознак – найважливіша задача інформатики. Придумано багато методів її розв’язання, розроблені дотепні алгоритми, але обсяг інформації все зростає, і потрібні усе більш швидкі алгоритми пошуку.

Методи пошуку різні для різних способів зберігання інформації. Ми розглянемо тільки пошук інформації, яка знаходиться в масиві.

Найпростіший спосіб пошуку інформації в масиві – просто перебирати всі його елементи із самого початку, поки не

зустрінеться потрібний елемент або поки не закінчиться масив. Такий метод називається лінійним пошуком.

Нехай для простоти в нас є цілочисельний масив:

```
int a[LEN];
```

Він заповнений якимись числами. Нам треба відшукати в цьому масиві задане число `x` і повернути індекс `m` елемента, що містить `x`, або число `-1`, якщо такого елемента в масиві немає. Пошук можна організувати так:

```
int m=0;
while(m<LEN&&a[m]!=x) m++;
if (m==LEN) m=-1;
```

Такий алгоритм зажадає в середньому `LEN/2` ітерацій циклу.

Припустимо, що масив `a` впорядкований за зростанням або убуванням чисел. Тоді можна організувати бінарний пошук. Порівняємо число `x` із середнім за величиною елементом масиву. Це буде елемент `a[LEN/2]`. Якщо `x` менше цього елемента, то його треба шукати в першій половині масиву, якщо більше – в другій половині. Вибираємо потрібну половину й повторюємо з нею попередні дії.

Саме так ми відшукуємо телефонний номер у довіднику за прізвищем абонента: розкриваємо довідник приблизно посередині, дивимося, де опиниться прізвище – ближче до початку довідника або ближче до кінця, – відкриваємо довідник посередині його першої або другої половини й т. д.

Як можна помітити, алгоритм бінарного пошуку схожий на метод бісекції для знаходження кореня нелінійного рівняння, наведений у лістингу 5. Для його реалізації введемо менший і більший індекси `left` і `right` поточного підмасиву. Спочатку ми розглядаємо весь масив, отже, їхні початкові значення будуть `0` і `LEN`. Індекс шуканого елемента `m` встановлюємо між ними: `left <= m < right`.

Після кожного порівняння середнього елемента підмасиву `a[m]` зі значенням `x` залежно від вибору або індекс `left` стає рівним `m+1`, або індекс `right` робиться рівним `m`. Тим самим підмасив зменшується вдвічі. Цикл повторюється доти, поки

індекс left не зрівняється з індексом right. Після нього потрібна додаткова перевірка. Все це реалізовано в лістингу 11.

Лістинг 11. Бінарний пошук у масиві

```
#include  <stdio.h>
#define LEN 10
main() {
int a[]={-10,-8,-3,0,2,4,7,11,13,17};
int x=11;
int left=0, right=LEN, m=-1;
while (left<right){
m = (left+right)/2;
if (x<=a[m]) right=m;
else left=m+1;
 }
if (right==LEN||a[right]!=x) m=-1;
printf("x=%d,a[%d]=%d\n",x,m,(m!=-1)?a[m]:
9999);
}
```

Бінарний пошук набагато швидше простого лінійного пошуку. У середньому він вимагає такого числа повторень циклу, що дорівнює двійковому логарифму величини LEN. Якщо в масиві 100000 елементів, то ми одержуємо середній виграш майже в 3000 разів! Зрозуміло, що лінійний пошук ні на що не придатний, але для застосування бінарного пошуку масив треба попередньо відсортувати.

Вправи

1 Перевірте, чи буде працювати алгоритм бінарного пошуку, записаний у лістингу 11, якщо в масиві кілька разів зустрічається шуканий елемент.

2 Напишіть програму бінарного пошуку в упорядкованому за алфавітом масиві рядків.

20 МЕТОДИ СОРТУВАННЯ МАСИВУ

Існує кілька десятків способів сортування, і їхнє число увесь час зростає. Постійно виходять цілі книги, присвячені тільки методам сортування. Познайомимося з найпростішими алгоритмами.

Всі пропоновані алгоритми виконують сортування на місці, в області пам'яті, зайнятій масивом. Це обов'язкова умова для більших масивів, що займають майже всю доступну оперативну пам'ять.

20.1 Сортування вставленням

Метод сортування вставленням застосовується в тих випадках, коли масив тільки що створений або триває його створення, і його треба заповнювати так, щоб після вставлення кожного нового елемента збереглася його впорядкованість. Для цього здійснюється пошук підходящого для вставлення місця у вже заповненій частині масиву `a[0]...a[m]`. Пошук триває від кінця до початку масиву. Місце для нового елемента звільняється шляхом зсуву елементів масиву до кінця масиву, як показано в лістингу 12.

Лістинг 12. Сортування масиву простим вставленням

```
#include <stdio.h>
#define LEN 10
main(){
int a[LEN];
int m=0, x;
scanf("%d", &x);
while (m<LEN-1 && x != 9999){
int t=m;
while (t>=0&&x<a[t]){
a[t+1]=a[t];
t--;
}
a[t+1]=x;
m++;
scanf("%d", &x);
 }
```

```
}
```

У циклі while лістингу 12 виконується лінійний пошук. Оскільки початок масиву вже впорядковано, краще організувати бінарний пошук, як зроблено в лістингу 13.

Лістинг 13. Сортування масиву бінарним вставленням

```
#include <stdio.h>
#define LEN 10
main() {
int a[LEN];
int length=0, m, x;
scanf("%d", &x);
while (length<LEN&&x!=9999){
int left=0, right=length;
while (left<right){
   m=(left+right)/2;
   if (x<=a[m]) right=m;
   else left=m+1;
 }
int k;
for (k=length; k>=right; k-k--)
       a[k + 1] = a[k];
 a[right]=x;
 length++;
 scanf("%d", &x);
}
}
```

Чи дасть бінарне вставлення великий виграш у цьому алгоритмі? Спірне питання, оскільки модифікація стосується тільки пошуку, а переміщення елементів, як і раніше, лінійне, для нього організований додатковий цикл.

20.2 Сортування вибором

Метод сортування вибором застосовується в тих випадках, коли масив не впорядкований, але його елементи треба виводити в зростаючому порядку. У такому випадку ми відшукуємо найменший елемент, виводимо його й на його місце ставимо яке-

небудь значення, свідомо більше всіх елементів масиву. Потім знову виводимо найменший елемент і т. д.

Такий метод придатний тільки для виводу масиву, оскільки масив знищується, його елементи поступово заміняються якимось більшим значенням. Проте його можна застосувати й для сортування на місці, якщо не виводити найменший елемент, а поміняти його місцями з першим елементом. При наступному пошуку найменшого елемента перший елемент масиву вже не треба брати до уваги. Так зроблено в лістингу 14.

Лістинг 14. Сортування масиву вибором

```
#include <stdio.h>
#define LEN 10
main(){
int a[]={15, 12, 5, 7, 0, -2, 4, 8, -3, 10};
int k, m;
for (k=0; k<LEN-1; k++){
  int ind=k;
  for (m=k+1; m<LEN; m++)
  if (a[m]<a[ind]) ind=m;
int t=a[k]; a[k]=a[ind]; a[ind]=t;
}
}
```

Сортування вибором дуже просто програмується, але воно не ефективне, оскільки вимагає багато порівнянь і перестановок.

20.3 Сортування методом "пухирця"

Наведемо ще один малоефективний, але характерний метод сортування масиву – метод *простого обміну,* частіше називаний методом “*пухирця'”* (buble method). Він практично не застосовується на практиці, але обов'язково вивчається всіма програмістами, оскільки простий й характерний для цілої групи методів сортування – сортувань обміном (лістинг 15).

Даний метод полягає в порівнянні пари сусідніх елементів і заміні їх місцями, якщо елементи утворюють інверсію. Після

цього порівнюється наступна пара й т. д. Перегляд усього масиву доводиться робити кілька разів, як показує наступний приклад.

Вихідний масив:

```
7, 5, 4, 8, 2, 3
```

Перший перегляд масиву й перестановки його елементів:

```
7, 5, 4, 8, 2, 3
5, 7, 4, 8, 2, 3
5, 4, 7, 8, 2, 3
5, 4, 7, 2, 8, 3
5, 4, 7, 2, 3, 8
```

Другий перегляд і перестановки:

```
5, 4, 7, 2, 3, 8
4, 5, 7, 2, 3, 8
4, 5, 2, 7, 3, 8
4, 5, 2, 3, 7, 8
```

Третій перегляд і перестановки:

```
4, 5, 2, 3, 7, 8
4, 2, 5, 3, 7, 8
4, 2, 3, 5, 7, 8
```

Четвертий перегляд і перестановки:

```
4, 2, 3, 5, 7, 8
2, 4, 3, 5, 7, 8
2, 3, 4, 5, 7, 8
```

Як бачите, навіть такий простий масив зажадав чотири перегляди. При першому перегляді найбільший елемент 8 сплив як пухирець на своє місце. При другому перегляді сплив наступний за величиною елемент 7 і т. д. Тому при кожному наступному перегляді ми можемо обмежитися тільки першими елементами масиву, останні елементи встали на свої місця при перших переглядах масиву.

Лістинг 15. Сортування масиву методом "пухирця". Перший варіант

```
#include <stdio.h>
#define LEN 10
main(){
  int a[]={15,12,5,7,0,-2,4,8,-3,10};
  int k, m;
  for (k=LEN-1; k>1; k--)
    for (m=0; m<k; m++)
      if (a[m]>a[m+1]){
         int t=a[m];a[m]=a[m+1];a[m+1]=t;
         }
  }
```

Цей алгоритм можна поліпшити. Кількість переглядів масиву в ньому не залежить від його впорядкованості. Навіть якщо масив споконвічно впорядкований, алгоритм лістингу 15 буде переглядати його len-2 рази. Введемо в алгоритм прапор упорядкованості – цілочисельну змінну `flag`, що прийме значення 1, якщо масив уже впорядкований, тобто при перегляді масиву не було зроблено жодної перестановки.

Лістинг 16. Сортування масиву методом "пухирця". Другий варіант

```
#include <stdio.h>
#define LEN 10
main(){
int a[]={15,12,5,7,0,-2,4,8,-3,10};
int k, m, flag = 0;
for (k = LEN -1; k > 1; k--){
int flag = 1;
for (m = 0; m < k; m++)
  if (a[m] > a[m + 1]){
    int t=a[m]; a[m]=a[m+1];a[m+1]=t;
    flag = 0;
   }
  if (flag) return;
 }
}
```

20.4 Метод просіювання

Метод "пухирця" можна поліпшити так званим просіюванням масиву. Воно полягає в тому, що після кожної перестановки ми продовжуємо переставляти менший елемент із усіма попередніми елементами масиву, поки він не встане на своє місце. Одні "пухирці" спливають, інші тонуть. Подивимося, як це відбувається на тім же прикладі.

```
7, 5, 4, 8, 2, 3
5, 7, 4, 8, 2, 3
5, 4, 7, 8, 2, 3
4, 5, 7, 8, 2, 3
4, 5, 7, 2, 8, 3
4, 5, 2, 7, 8, 3
4, 2, 5, 7, 8, 3
2, 4, 5, 7, 8, 3
2, 4, 5, 7, 3, 8
2, 4, 5, 3, 7, 8
2, 4, 3, 5, 7, 8
2, 3, 4, 5, 7, 8
```

Як бачите, у методі просіювання робиться тільки один перегляд масиву. Для реалізації методу просіювання після перестановки елементів масиву треба вставити ще один цикл перестановок з попередніми елементами, як зроблено в лістингу 17.

Лістинг 17. Сортування масиву методом просіювання

```
#include <stdio.h>
#define LEN 10
main(){
  int a[] = {15, 12, 5, 7, 0, -2, 4, 8, -3,
10};
  int m;
  for (m = 0; m < LEN - 1; m++)
    if (a[m] > a[m + 1]){
      int t = a[m]; a[m] = a[m + 1]; a[m +
1] = t;
      int s = m;
      while (s > 0 && a[s] < a[s - 1]){
```

```
            t = a[s]; a[s] = a[s - 1]; a[s - 1]
= t;
            s--;
           }
        }
    }
```

20.5 Метод Шелла

У методах "пухирця" і просіювання переставляються сусідні елементи масиву. Сортування істотно прискорюється, якщо робити порівняння й перестановку далеко віддалених один від одного елементів. Правда, при цьому доведеться робити кілька переглядів масиву, але загальна швидкість алгоритму при цьому зростає. У методі Шелла рекомендується спочатку порівнювати елементи `a[k]` і `a[k + h]`, що відстоять один від одного на половину довжини масиву: `h = len/2`. Потім крок зменшується наполовину: `h /= 2`. В останньому перегляді `h = 1`, тобто виконується звичайне просіювання. Цей метод представлений у лістингу 18.

Лістинг 18. Сортування масиву методом Шелла

```
    #include <stdio.h>
    #define LEN 10
    main(){
     int a[] = {15, 12, 5, 7, 0, -2, 4, 8, -3,
10};
     int m, h = LEN/2;
     while (h >= 1){
       for (m = 0; m < LEN - h; m++)
         if (a[m] > a[m + h]){
          int t = a[m]; a[m] = a[m + h] ; a [m
+ h] = t,-
          int s = m;
          while (s > h && a[s] < a[s - h]){
             t = a[s]; a[s] = a[s - h] ,- a[s -
h] = t;
             s--;
          }
```

```
      }
    h/=2;
   }
}
```

20.6 Огляд методів сортування

Перераховані методи сортування – це тільки невелика їхня частина. Ми займалися тільки сортуваннями обміном елементів масиву. Крім них, є ще дві великі групи методів сортувань: сортування за допомогою дерев і пірамідальні сортування.

Навіть серед сортувань обміном ми не розглянули найшвидший метод, що так і називається – швидке сортування (quick sort). Цей метод заснований на рекурсії, і ми його розглянемо окремо і пізніше.

Контрольні питання

1 Чи можна в мові С використовувати негативні індекси масивів?

2 Яке максимальне число індексів у масивів, що задаються в мові С?

3 Що буде, якщо індекс масиву виявився речовинним, а не цілим значенням?

4 Чи можна як індекси масиву використовувати символи?

5 Чим небезпечний висячий покажчик?

6 У чому різниця між звичайним покажчиком та іменем масиву?

7 Чому при визначенні покажчика треба обов'язково задавати тип?

8 Де в мові С з'являються нетипізовані покажчики і як потрібно з ними працювати?

9 Що таке рядок у мові С?

10 Що підставляє компілятор на місце рядкової константи?

11 У якій області пам'яті зберігають рядкові константи деякі компілятори?

12 Скільки часу зберігається пам'ять, виділена статично?

13 Скільки часу зберігається пам'ять, виділена динамічно?

14 Як звільнити пам'ять, виділену динамічно?

21 РОБОТА З ФАЙЛАМИ

Часто введення й виведення програми пов'язується із зовнішніми пристроями: жорсткими й гнучкими дисками, принтерами, сканерами, картами флеш-пам’яті. На цих пристроях інформація звичайно зберігається у файлах, з якими програма повинна вміти працювати: відшукувати на диску, читати їх, записувати в них інформацію, створювати нові.

У стандартну поставку мови С входить велика бібліотека функцій введення-виведення інформації. Вона описана в головному файлі `stdio.h (standard input-output)`. Ми вже використовували функцію форматованого введення `scanf()` і функцію форматованого виводу `printf()` із цієї бібліотеки.

Функції, що входять до бібліотеки введення-виведення, розглядають файл як послідовність байтів. Байти пронумеровані починаючи з 0. Функції дозволяють переміщати покажчик по файлу в обох напрямках і безпосередньо звертатися до будь-якого байта. Номер поточного байта у файлі відслідковується при всіх операціях з файлом. Він називається поточною позицією (`current position`) файла. Функції введення-виведення можуть прочитати з файла або записати у файл будь-яку кількість байтів починаючи з поточної позиції, після чого поточна позиція збільшується на оброблене число байтів.

Перед роботою з файлом він повинен бути відкритий (`open`). Операція відкриття містить у собі пошук каталога, у якому описаний файл, пошук файла на диску за інформацією, отриманою з каталога, перевірку блоків файла, створення в оперативній пам'яті спеціальної структури й заповнення її інформацією про файл. Тип цієї структури `file`, він описаний у головному файлі `stdio.h`. Після відкриття файла можна зайнятися пошуком (`seek`) потрібного байта у файлі, як говорять за аналогією з магнітною стрічкою, перемотувати файл.

Знайшовши початок потрібної інформації, можна прочитати (`read`) її або записати (`write`) у потрібне місце нову інформацію. При читанні файла інформація спочатку записується в спеціально виділену область оперативної пам'яті – буфер

(buffer) введення, а потім забирається звідти й записується в змінні й масиви програми, що читає файл. Те саме відбувається й при виведенні у файл. Інформація із програми спочатку накопичується в буфері виведення, а потім записується на диск. Буферизація введення-виведення значно прискорює операції з файлом. Для ще більшого прискорення роботи програма, як правило, створює не один, а кілька буферів введення-виведення. У той час як один буфер заповнюється, програма забирає інформацію з іншого.

Нарешті, по закінченні роботи з файлом його треба закрити (close). Операція закриття скидає вміст буферів у файл, обновляє інформацію про файл у каталозі й звільняє оперативну пам'ять, зайняту буферами й структурою типу FILE. Під час завершення програми операційна система автоматично закриває всі відкриті програмою файли, але щоб не засмічувати оперативну пам'ять, краще закривати файли відразу ж після роботи з ними.

Отже, вся робота з файлами полягає всього в декількох діях: "відкрити", "відшукати", "прочитати/записати" і "закрити". Кожна дія виконується відповідною функцією бібліотеки введення-виведення. Розглянемо їх докладніше.

21.1 Функції буферизованого введення-виведення

Відкриття файлу виконує функція fopen(). Їй передаються два аргументи: ім'я файла у вигляді покажчика на рядок і ще один рядок, що задає режим відкриття файла. Функція серед інших дій визначає структуру типу FILE і повертає покажчик на неї. Цей покажчик потім використовують інші функції, щоб одержати інформацію про відкритий файл. Якщо файл не вдалося відкрити, то вертається константа NULL. Наприклад:

```
    #include <stdio.h>
    FILE *f1 = fopen("C:\\myworks\\
myfile1.txt", "r");
    FILE *f2 = fopen("/home/khabib/myworks/
myfile2.txt", "w");
```

У першій функції fopen() повне ім'я файла написане за правилами операційної системи MS-DOS. Каталоги розділяються

зворотною похилою рискою, що доводиться подвоювати, оскільки в мові С у зворотної похилої риски спеціальне значення. У другій функції повне ім'я файла написане за правилами операційних систем UNIX.

У класичних операційних системах файл відкривається в одному із двох режимів: на читання або на запис. Файл, відкритий на читання, можна переглядати, зчитувати з нього будь-яку інформацію, але його не можна змінювати.

Файл, відкритий на запис, насамперед очищається. Якщо в ньому містилася якась інформація, то вона видаляється, як говорять за аналогією з магнітною стрічкою, стирається. Потім відбувається запис у порожній файл. У функції `fopen`() режим читання задається рядком `r`, що складається з одного символу, а режим запису – рядком `w`.

Пам’ятайте – файл, відкритий на запис, стирається!

Якщо ви захочете прочитати тільки що записану у файл інформацію, то файл спочатку треба буде закрити, а тільки потім відкрити на читання. Якщо файл, що відкривається на читання, не існує, то видається повідомлення про помилку й виконання програми припиняється. Якщо не існує файл, що відкривається на запис, то він створюється.

Оскільки операція запису не дозволяє доповнити існуючий файл новою інформацією, був введений ще один режим відкриття: відкриття файла на дозаписування. У цьому режимі файл відкривається на запис і відразу перемотується в кінець, так що запис йде в кінець файла. Колишній вміст файла змінити неможливо, навіть перемотавши файл на початок. У функції `fopen`() режим дозаписування задається рядком `a (append)`.

У сучасних операційних системах файл, як правило, відкривається відразу й на читання, і на запис. Його можна перемотувати, читати й записувати, не закриваючи й не відкриваючи знову. Проте стандарт мови С рекомендує між операціями читання й запису робити перемотування функціями `fseek`() або `rewind`(). Якщо насправді не треба переміщатися по файлу, то можна зробити несправжнє перемотування, перемотавши файл на нуль байтів. У функції `fopen`() розширені режими відкриття файла позначаються плюсом: `r+, w+, a+`. Не забувайте, що в режимі `w+` вміст файла, якщо він був не

пустим, зітреться, а в режимі a+ колишній вміст файла змінити не можна.

Деякі операційні системи розрізняють текстові файли, у яких кожний байт вважається символом, і бінарні файли, що містять зображення, звуки або машинні команди. У таких системах режим відкриття уточнюється параметром t (text) або b (binary), наприклад rb. Якщо такого уточнення немає, то за замовчуванням розуміється текстовий файл. У текстовому файлі символи переведення рядка \n і повернення каретки \r обробляються по-різному в різних операційних системах, тому треба обережно працювати з файлом, відкритим як текстовий файл.

Отже, у функції fopen() можна задати один із шести режимів відкриття файла. Зверніть увагу, що режим задається рядком, а не символом, тобто треба завжди записувати лапки, а не апострофи, навіть якщо режим записується однією буквою. Після відкриття можна переміститися по файлу за допомогою функції fseek(). Їй передаються три аргументи: покажчик на файл, зсув (offset) – число байтів, на які треба пересунутися по файлу, і напрямок переміщення, що задається числом 0, 1 або 2.

Якщо напрямок зазначений числом 0, то переміщення йде від початку файла. У цьому випадку зсув має бути невід'ємним. Якщо напрямок заданий числом 1, то переміщення починається від поточної позиції у файлі. У цьому випадку зсув може бути й невід'ємним, і від'ємним. Нарешті, якщо напрямок переміщення зазначений числом 2, то воно йде від кінця файла до його початку, і зсув повинний бути від'ємним. У багатьох системах програмування у файлі stdio.h визначені константи seek_set, seek_cur і seek_end, що відповідають числам 0, 1 і 2.

Зсув задається довгим цілим числом типу long. Оскільки компілятор не робить перетворень типу, його треба дотримуватися навіть у константах. Наприклад:

```
    #include <stdio.h>
    FILE *f;
    if
((f=fopen("/home/khabib/myworks/myfile.txt","r+
"))!=NULL){
```

```
    fseek(f, 200L, 0);/*Перемотування на 200
байтів від початку файла.*/
    fseek(f, 200L, SEEK_SET);/*Те саме.*/
    fseek(f, -100L, SEEK_CUR);/*Перемотування
на 100 байтів від поточної позиції до початку
файла.*/
    fseekff, 0L, SEEK_END);/*Перемотування в
кінець файла*/
    fseek(f, 0L, SEEK_SET);/*Перемотування в
початок файла*/
    }
```

Функція `fseek()` повертає 0 у випадку вдалого переміщення й ненульове значення при помилці, наприклад при спробі перемістится по ще не відкритому файлу. При роботі з текстовими файлами функція `fseek()` може виконати неправильне перемотування, викликане різною інтерпретацією символів переведення рядка й повернення каретки.

Для зручності роботи з файлами в мову С введена функція `rewind()`, що перемотує файл на початок. У неї один аргумент – покажчик на відкритий файл, і немає значення, що повертається.

Поточна позиція файла зберігається в структурі типу `FILE`, створеній при відкритті файла. Вона повертається функцією `ftell()` у вигляді числа типу `long`. Після того як установлена правильна позиція у файлі, можна читати або записувати інформацію. Читання здійснюється функцією `fread()`. У неї чотири аргументи. Перший аргумент – покажчик типу `(char*)` на буфер, у який буде записана прочитана інформація. Другий аргумент – кількість байтів в одній порції інформації, що читається, третій аргумент – кількість таких порцій. Нарешті, четвертий аргумент – покажчик на структуру типу `FILE`, створену при відкритті файла. Наприклад:

```
    #include <stdio.h>
    FILE *f; char buf[4096];
    if
((f=fopen(Vhome/khabib/myworks/myfile.txt","r+"
))!=NULL)
    freadtbuf, 4096, 1, f);
```

Виникає питання, що краще: прочитати один раз 4096 байтів, як написано вище, або 4096 разів по одному байту, як написано нижче?

```
fread(buf, 1, 4096, f);
```

Відповідь проста й очевидна: оскільки операційна система читає й записує інформацію блоками, краще переносити за один раз цілу порцію байтів. Тут допомагає операція `sizeof`(), що обчислює довжину типу або масиву. Наприклад:

```
fread(buf, sizeof(buf), 1, f);
```

Але найкраще читати інформацію блоками операційної системи. Розмір блока зберігається в константі `bufsiz`, певної у файлі `stdio.h`. Наведений приклад можна записати так:

```
#include <stdio.h>
FILE *f;
char buf[BUFSIZ];
if((f=fopen(Vhome/khabib/myworks/
myfile.txt'','r+"))!=NULL)
fread(buf, BUFSIZ, 1, f);
```

Функція запису інформації у файл `fwrite`() виглядає так само, але переносить інформацію в протилежному напрямку – з буфера `buf` у файл `f`.

```
#include <stdio.h>
FILE *f;
char buf[] = "Інформація, записувана у
файл.";
if
((f=fopen("/home/khabib/myworks/myfile.txt","w+
"))!=NULL)
    fwrite(buf,sizeof(buf),1,f);
```

Функції `fread`() і `fwrite`() повертають число фактично оброблених повних порцій або 0, якщо читання чи запис не вдалися, наприклад зустрівся кінець файла. Після читання або запису поточна позиція файла збільшується на число прочитаних або записаних байтів. Перевірити закінчення файла можна функцією `feof`(), що повертає ненульове значення, якщо

попереднє читання дійшло до кінця файла, і 0, якщо кінець файла ще не досягнуто.

Після роботи з файлом його треба закрити функцією `fclose()`. При закритті файла буфер очищається, але до цього може статися, що остання порція інформації залишиться в буфері й не буде записана у файл. Для того щоб уникнути цієї неприємності, треба примусово очистити буфер, скинувши його вміст у файл. Цю роботу виконує функція `fflush()`.

Повний цикл роботи з файлами наведений у лістингу 19.

Лістинг 19. Функція введення-виведення низького рівня

```
#include <stdio.h>
main(){
  char but[BUFSIZ];
  FILE *fr, *fw;
  if ( (fr = fopen("myoldfile.txt", "r")) == NULL){
      printf("Не вдається відкрити існуючий файл.\n");
      return -1;
  }
  if ( (fw = fopen("mynewfile.txt", "w")) == NULL){
      printf("Не вдається відкрити новий файл\n");
      return -1;
  }
  while (!feof(fr)){
     fread(buf, BUFSIZ, 1, fr) ;
     /* Робота з буфером buf */
     fwrite(buf, BUFSIZ, 1, fw) ;
   }
   fclose(fr);
   fclose(fw);
  }
```

Вправи

1 Прочитайте початок якого-небудь із наявних у вас на диску файлів у буфер і виведіть зміст буфера на екран. Подивіться, у якому вигляді зберігаються байти цього файла.

2 Запишіть масив чисел у файл. Прочитайте цей файл і відновіть масив.

3 В операційних системах сімейства UNIX перші чотири байти більшості типів файлів – це "magic number" – число, що показує тип файла. Напишіть функцію, що повертає перші чотири байти файла, заданого аргументом функції.

4 В операційній системі MS-DOS перші два байти EXE-файлів дорівнюють 0100110101011010. Це ініціали "MZ" творця EXE-формату Марка Збиковськи (Mark Zbikowski). Напишіть функцію, що перевіряє ці байти у файлі, заданому її аргументом, і що повертає 1, якщо це EXE-файл, і 0 в іншому випадку.

22 ПОТОКИ ВВЕДЕННЯ-ВИВЕДЕННЯ

З комп'ютером зв'язана велика кількість зовнішніх пристроїв: принтери, сканери, компакт-дисководи, карти флеш-пам'яті, цифрові фотоапарати й відеокамери. Обмін інформацією з кожним з них має свої особливості, які доводиться враховувати при програмуванні введення-виведення. Більше того, навіть у різних моделей того самого пристрою буває різниця в передачі інформації. Вся історія програмування пронизана прагненням згладити цю різницю. Поняття файла – одна з великих удач на цьому шляху. Ми звертаємося до файла на ім’я, не думаючи про те, на якому циліндрі диска він розташований. Ми розглядаємо файл як послідовність байтів, не знаючи про те, які блоки диска він займає, де розташовані ці блоки і як записані байти в цих блоках.

Наступним кроком на шляху уніфікації роботи із зовнішніми пристроями стало поняття потоку (`stream`) інформації. Введення інформації розглядається як вхідний потік байтів, "що втікає" звідкись у комп'ютер (`input stream`), виведення – як вихідний потік байтів, "що витікає" з комп'ютера (`output stream`). При цьому нам цілком не важливо, звідки надходить потік інформації й куди він іде. Це справа операційної

системи. Ми пишемо функції введення-виведення, беручи вихідну інформацію із вхідного потоку й відправляючи результати у вихідний потік. Перед виконанням програми операційна система зв'язує потоки з конкретними пристроями. Вона може перепризначити потік: зараз інформація виводиться на принтер, через якийсь час вона піде у файл або буде спрямована в мережу.

Всі стандартні функції введення-виведення мови C розраховані насправді не на роботу з файлами, а на роботу з потоками. Покажчик на структуру типу FILE – це створюваний при відкритті файла потік. На іншому його кінці може виявитися не диск, а зовсім інший пристрій, аби тільки воно вміло працювати з файлами за їхніми іменами.

Перед початком роботи кожної програми для неї автоматично відкриваються три стандартних потоки: вхідний потік з номером 0 та іменем `stdin`, вихідний потік з номером 1 та іменем `stdout`, вихідний потік з номером 2 та іменем `stderr`. Вхідний потік відразу ж зв'язується із клавіатурою, а вихідні потоки – з екраном дисплея. Потім ці пристрої можна перепризначити.

Вихідний потік `stderr` розрахований на передачу програмою повідомлень про свою роботу. Найчастіше це попередження й повідомлення про помилки, звідки й відбулася назва "`standard error stream`". Через те що обидва стандартних вихідних потоки спрямовані на той самий пристрій, вони часто змішуються, що ви, напевно, могли спостерігати на консолі свого комп'ютера: серед результатів вашої програми зненацька починають з'являтися малозрозумілі повідомлення про якісь помилки. Щоб не відбувалося такого змішання, багато програм, особливо сервери, направляють потік `stderr` у який-небудь журнальний (`log`) файл, у якому накопичується повний звіт про роботу програми. Всі адміністратори комп'ютерних систем добре знають про те, що треба регулярно переглядати журнальні файли й час від часу очищати їх.

Функція `scanf`(), що ми користувалися раніше, `розрахована` на введення зі стандартного потоку `stdin`, функція `printf`() – на виведення у стандартний потік `stdout`.

Функції fread() і fwrite() можна зв'язати зі стандартними потоками:

```
fread(buf, BUFSIZ, 1, stdin);
fwrite(bufl, sizeof(bufl), 1, stdout);
fwrite(bufl, sizeof(bufl), 1, stderr);
```

При цьому стандартні потоки відкривати не треба, вони вже відкриті.

Потоки stdin, stdout і stderr – це константи, вони не можуть привласнити інший потік. Перепризначити стандартний потік можна засобами операційної системи при запуску здійсненного файла:

```
$./mycoolprog >myoutputfile>mylogfile
```

Перед виконанням mycoolprog стандартне виведення stdout буде спрямований у файл myoutputfile, а стандартний потік повідомлень stderr буде спрямований у файл mylogfile. Оскільки файли myoutputfile і mylogfile відкриваються на запис, вони попередньо будуть очищені.

Цей спосіб дуже розповсюджений в операційних системах сімейства UNIX. Програма для них дуже часто пишеться розраховуючи на стандартні потоки введення-виведення, а за необхідності потоки перенаправляються операційною системою.

23 ФОРМАТОВАНЕ ВВЕДЕННЯ-ВИВЕДЕННЯ

У стандартній бібліотеці введення-виведення є дві функції, аналогічні функціям scanf() і printf(), але розраховані не тільки на стандартні потоки. Це функції fscanf() і fprintf(). Попередньо відкритий потік у них вказується першим аргументом, а інші аргументи такі самі, як у функцій scanf() і printf().

Строго говорячи, все виглядає так: функція scanf ("format", arguments,...) - це окремий випадок функції fscanf(), а саме: fscanf(stdin, "format", arguments,...). А функція printf("format",

arguments, . . .) – це функція fprintf (stdout, "format", arguments, . . .).

Функцією fprintf() дуже часто користуються для запису повідомлень про помилки, направляючи їх у потік stderr:

```
if ( (f = fopen("myfile", "r")) == NULL) {
fprintf(stderr, "Неможливо відкрити файл\
n");
return -1;
}
```

Функцією fscanf() користуються рідше, тому що вона вимагає ретельного запису вхідного потоку. Будь-яка помилка у вхідному потоці призведе до непередбачених результатів у її роботі.

Ще одна корисна функція із цього сімейства заносить інформацію не у вихідний потік, а в рядок символів. Це sprintf(). Першим її аргументом служить не покажчик на структуру типу FILE, а покажчик на рядок типу (char*). Ця функція зручна для запису чисел у вигляді рядків, що складаються із цифр, десяткових точок, знаків плюс і мінус. Наприклад:

```
shar s[100] ;
sprintf(s, "Сьогодні %d градусів.",
temperature);
```

У бібліотеці є й функція fscanf() форматованого введення з рядка символів за зазначеними форматами у змінні, адреси яких зазначені наступними аргументами. Приклад:

```
int k;
char s[10] ;
sscanf("126 ламп", "%d%s", &k, s);
```

23.1 Інші функції введення-виведення

Ми розглянули функції введення-виведення найнижчого рівня fread() і fwrite(), що являють собою введення-виведення даних у вигляді простого потоку байтів. Ці функції не

цікавляться тим, яку інформацію несуть байти, числа це, символи або машинні команди. Вони не можуть перевірити правильність переданої інформації, не можуть відстежити перекручені або загублені байти.

З іншого боку, ми користувалися функціями введення-виведення найвищого рівня `fscanf()` і `fprintf()`, що являють собою інформацію у вигляді значень простих типів даних. Вони перевіряють дані, що вводяться або виведені, намагаються їх перетворити до зазначеного типу й повідомляють про невдалі спроби.

Між цими двома парами функцій розташовуються функції введення-виведення, що передають окремі символи або рядки символів.

Функція із заголовком `int fgetc(FILE *stream)` уводить у програму один символ з потоку, заданого своїм аргументом, повертаючи його як тип `int`. Тип `int`, а не `char` обраний тому, що у випадку невдачі функція повертає константу `eof`, що дорівнює -1, що не відповідає типу `char`. Сам символ у форматі типу `unsigned char` буде займати молодший байт значення, що повертається.

```
FILE *f;
int c;
if ((f = fopen("myfile","r+")==NULL){
fprintf(stderr,"Помилка відкриття файла.");
   return -1;
}
if ((c=fgetc(f)==EOF) {
 fprintf(stderr,"Помилка читання файла.");
   return -1;
}
char ch = (char)c;
```

Функція із заголовком `int fputc(char c, FILE *stream)` виводить символ `c`, зазначений першим аргументом у вихідний потік `stream`, заданий другим аргументом. Функція повертає в молодшому байті виведене значення або `eof` у випадку невдачі.

У лістингу 20 наведена програма посимвольного копіювання файлів. Вона зручна для роботи із символьними пристроями введення-виведення. Імена файлів уводяться в командному рядку при запуску програми на виконання.

Лістинг 20. Посимвольне введення-виведення

```
#include <stdio.h>
main(int argc, char **argv){
FILE *in, *out;
int ch;
if (argc != 3){
  printf("Usage: mycopy source target\n");
  return -1;
}
if ( (in = fopen(argv[l], "r")) == NULL){
  printf("Не вдається відкрити файл, що
копіюється.\n");
  return -1;
}
if((out=fopen(argv[2],"w"))==NULL){
  printf("Не вдається відкрити новий файл.\
n");
  return -1; )
}
while((ch=fgetc(in))!=EOF)fputc(ch,out);
fclose(in); fclose(out);
}
```

Функції `int getchar()` і `int putchar(char c)` роблять те саме, але вони жорстко пов'язані зі стандартними потоками `stdin` і `stdout` відповідно. Виклик `getchar()` еквівалентний виклику `fgetc(stdin)`, виклик `putchar(c)` еквівалентний виклику `fputc(c, stdout)`.

От як можна організувати луну-відображення на екрані символів, що вводяться із клавіатури:

```
int c;
while ((c = getchar()) != EOF) putchar(c);
```

Оскільки stdout – буферизований потік, причому його буфер очищається тільки після свого заповнення або після влучення в нього символу переведення рядка, символи будуть відображатися на екрані тільки після натискання клавіші <Enter>. Для закінчення введення із клавіатури треба натиснути комбінацію клавіш <Ctrl>+<D>, увівши тим самим ознаку кінця файла.

Після роботи всіх цих функцій поточна позиція файла переміщається на один символ уперед.

Передача за один раз по одному символу часто виявляється недоцільною. У багатьох випадках зручніше вводити або виводити відразу цілий рядок. У такому випадку використовуються функції fgets() і fputs().

Функція із заголовком char *fgets(char *s, int size, FILE *stream) уводить максимум size - 1 символів із вхідного потоку stream і заносить їх у буфер, на який указує s, додаючи в кінець нуль-символ. Функція повертає покажчик на той же буфер s або NULL, якщо введення не вдалося.

Введення також закінчується при переповненні буфера й з появою в ньому символу переведення рядка \n.

Функція із заголовком int fputs(char *s, FILE *stream) виводить символи з буфера s до нуль-символу винятково у вихідний потік stream. Функція повертає код останнього виведеного символу, у випадку помилки вертається eof.

У лістингу 21 наведений ще один варіант копіювання файлів.

Лістинг 21. Порядкове введення-виведення

```
    #include <stdio.h>
    main () {
    FILE *infile, *outfile; char s[81];
if((infile=fopen("myoldfile.txt","r+"))==NULL)
  {
      printf("Не вдається відкрити файл, що
копіюється.\n");
```

```
      return -1;
    }
if((outfile=fopen("mynewfile.txt","w+"))==NULL)
{
      printf("Не вдається відкрити новий файл.\
n");
      return -1;
    }
    while(!feof(infile)){
      fgets(s, sizeof(s), infile);
      fputs(s, outfile);
    }
    fclose(infile);
    fclose(outfile);
    }
```

Для цих функцій є відповідна пара функцій введення-виведення рядків, пов'язаних зі стандартними потоками.

Функція із заголовком `char *gets(char *s)` вводить символи до символу переведення рядка винятково зі стандартного потоку `stdin` у буфер `s`, додаючи нуль-символ. Функція повертає покажчик на той самий буфер або `NULL`, якщо введення не вдалося.

Функція `int puts (char *s)` виводить символи з буфера s до нуль-символу винятково в стандартний вихідний потік `stdout`. Наприкінці додається символ переведення рядка \ n. У випадку помилки функція повертає eof.

Лістинг 22. Порядкове введення-виведення на консоль із перетворенням

```
    #include <stdio.h>
    #include <stdlib.h>
    main() {
      char name[81];
      char buf[81];
      puts("Введіть прізвище, ім'я та по
батькові:");
      gets(name);
```

```
  puts("Введіть суму внеску:");
  int deposit = atoi(gets(buf));
  printf("%s %d\n", name, deposit);
}
```

Функція із заголовком `int ungetc(int c, file *stream)` повертає один символ у буфер, пов'язаний із вхідним потоком `stream`. Цей символ можна буде прочитати наступним викликом функції `fgetc()`. Якщо значення `c` дорівнює `EOF`, то нічого не змінюється. Значення функції, що повертається, дорівнює `c` або `EOF`. Ця функція здається штучною, але її доводиться часто застосовувати при введенні. Наприклад, вводячи число й читаючи його цифри, ми повинні зупинитися в той момент, коли зустрівся символ, відмінний від цифри. Але для цього ми повинні прочитати цей символ. Він може належати наступному значенню, що вводиться, тому його не можна втрачати, а варто повернути в потік для наступного читання. Вхідний потік може бути недоступний, наприклад, він пов'язаний з мережею, тому повернення відбувається в буфер, пов'язаний з потоком.

От простий тренувальний приклад, що повертає символ "G" у порожній стандартний вхідний потік `stdin`, і забирає його звідти:

```
#include <stdio.h>
main () {
  char ch;
  ungetc('G', stdin);
  ch = getchar();
  putchar(ch);
}
```

У бібліотеці функцій введення-виведення є функції, що дозволяють працювати не тільки із вмістом файла, але й з файлом цілком.

Функція `int rename (const char *oldname, const char *newname)` змінює ім'я файла з `oldname` на ім'я `newname`. Вона повертає `0` при вдалому переіменуванні файла й `-1` у випадку невдачі. Наприклад:

```
rename ("C: \\works\\f1.txt", "D:\\ archive\\f1.arch");
```

Функція `int remove(const char *filename)` видаляє файл із ім'ям `filename`. Вона повертає `0` при вдалому видаленні файлу й `-1` у випадку невдачі.

Наприклад, `remove("f1.txt").`

Вправи

1 Напишіть функцію `www_encode(FILE *f)`, що читає текстовий файл, що заміняє спеціальні символи трьома символами `%xx`, де `xx` – шістнадцятковий код символу, і відправляє перетворений файл у стандартний вихідний потік.

2 Напишіть функцію `print_tab(char *filename, int tabsize)`, що читає текстовий файл із ім'ям `filename` і виводить його в стандартний вихідний потік із заміною символу горизонтальної табуляції на пробіли, число яких задає аргумент `tabsize`.

3 У текстовому файлі перебуває платіжна відомість із заголовком і таблицею із двома стовпцями: прізвищем з ініціалами й сумою виплати. Знайдіть середню виплату й виведіть її в стандартний вихідний потік.

24 ІНТЕРАКТИВНЕ ВВЕДЕННЯ-ВИВЕДЕННЯ

У багатьох системах програмування мовою С стандартна бібліотека функцій введення-виведення доповнена функціями, що працюють безпосередньо із клавіатурою й екраном дисплея. Такі функції описані у файлі `conio.h(console input-output).`

Функція `int getch()` вводить символ із клавіатури без луна-відображення його на екрані. Повертає код введеного символу. Введення не буферизується, тому процес введення не треба доповнювати натисканням клавіші `<Enter>` або очищенням буфера. Цю функцію часто використовують для припинення виконання програми до натискання якої-небудь

клавіші, наприклад, при перегляді на екрані великого обсягу інформації.

Функція `int getche()` вводить символ із клавіатури й відображає його на екрані. Вона повертає код введеного символу.

Функція `int putch(int ch)` виводить символ на екран дисплея. Вона повертає код виведеного символу.

```
#include <stdio.h>
#include <conio.h>
main(){
   char ch;
   while(!eof(stdin)){
      puts("Для продовження натисніть будь-
яку клавішу.");
      ch=getch();
      putch(ch);
      puts("Для закінчення натисніть
комбінацію клавіш Ctrl-D");
}
```

Функція `int ungetch(int c)` заносить символ `c` у буфер клавіатури й повертає його як результат своєї роботи. Наступна функція, що вводить символи із клавіатури, візьме саме цей символ, начебто б він був тільки що введений із клавіатури.

Функція `char cgets(char *s)` формує рядок `s`, розрахований на використання в різних обчислювальних системах. Елемент `s[0]` уже повинен містити максимальне число символів, що вводяться. В елемент `s[1]` функція заносить фактичне число уведених символів. Самі символи записуються, починаючи з елемента `s[2]`. Рядок завершується нуль-символом. Функція повертає покажчик на початок рядка `s`. Приклад:

```
char s[256];
s[0] = 253;
cgets(s);
printf("%s\n", s+2);
```

Функція `int cputs (const char *s)` виводить рядок `s` на екран дисплея. Її відмінність від функції `puts ()` тільки в тім, що вона не додає в кінець символ переведення рядка.

25 РОБОТА З КАТАЛОГОМ

Деякі системи програмування містять функції, що дозволяють працювати з каталогами. Ці функції описані у файлі `dir.h`. От деякі із цих функцій.

Функція `int mkdir(const char *dirname)` створює каталог з ім'ям `dirname` і повертає `0` при вдалому створенні каталогу й `-1` у випадку невдачі. Приклад її використання:

```
if (mkdir("C:\\inyworks\\iodir")){
  fprintf(stderr, "Не удалося створити каталог");
  return -1;
 }
```

Функція `int chdir(const char *dirname)` робить каталог `dirname` поточним каталогом. Вона повертає `0` при вдалій зміні каталогу й `-1` у випадку невдачі.

Функція `int rmdir(const char *dirname)` видаляє каталог `dirname`. Вона повертає `0` при вдалому видаленні каталогу й `-1` у випадку невдачі.

Контрольні питання

1 Що таке файл у мові С?

2 Яка різниця між бінарним і текстовим файлом?

3 Які операції можна робити з файлом за допомогою функцій мови С?

4 Що входить до процедури відкриття файла?

5 До чого призведе повторне відкриття вже відкритого файла?

6 Чи можуть кілька програм одночасно відкрити один і той самий файл?

7 Чи можна змінити вміст файла, відкритого в режимі "`a+`"?

8 Чи можна змінити вміст файла, відкритого в режимі "`r+`"?

9 Чи можна вийти за межі файла функцією `fseek()`?

10 Скільки байтів може прочитати функція `fread()` за один виклик?

11 У якому вигляді повинен бути записаний файл, що читається функцією `fscanf()`?

12 У якому вигляді запише інформацію у файл функція `fprintf()`?

13 Як очистити буфер виведення після запису всієї інформації у файл?

14 Якою функцією зручніше за все прочитати файл із текстом програми мовою С?

26 ЗАВДАННЯ НА ЗАКРІПЛЕННЯ МАТЕРІАЛУ

До розглянутого матеріалу додаються завдання з розроблення програм для закріплення отриманих знань і застосування їх при розробленні.

26.1 Геометрія

1 Із заданої множини точок на площині вибрати дві різні точки так, щоб кількості точок, що лежать по різні боки від прямої, яка проходить через ці дві точки, розрізнялися найменшим образом.

2 Визначити радіус і центр кола, на якому лежить найбільше число точок заданої на площині множини точок.

3 Задано множину ***M*** точок на площині. Визначити, чи правильно, що для кожної точки $A \in M$ існує точка $B \in M$ $(A \neq B)$ така, що не існує двох точок множини ***M,*** що лежать по різні боки від прямої ***AB***.

4 Визначити радіус і центр такого кола, що проходить хоча б через три різні точки заданої множини точок на площині, з мінімальною різницею кількостей точок, що лежать усередині й поза колом.

5 У множині точок на площині знайти пару точок з максимальною відстанню між ними.

6 Відстань між двома множинами точок – це відстань між найближче розташованими точками цих множин. Знайти відстань між двома заданими множинами точок на площині.

7 Багатокутник (не обов'язково опуклий) заданий на площині перерахуванням координат вершин у порядку обходу його границі. Визначити площу багатокутника.

8 Задано множину точок ***M*** у тривимірному просторі. Знайти таку з них, що куля заданого радіуса із центром у цій точці містить максимальне число точок з ***M***.

9 Задано множину прямих на площині (коефіцієнтами своїх рівнянь). Підрахувати кількість точок перетинання цих прямих.

10 У тривимірному просторі задана множина матеріальних точок. Знайти ту з них, що найближче розташована до центра ваги цієї множини.

11 У тривимірному просторі задана множина матеріальних точок. Кожна із точок з максимальною масою зникає, втрачаючи десяту частину своєї маси й роздаючи масу, що залишилася, нарівно всім іншим, більш «легким» точкам. Визначити сумарну масу множини матеріальних точок у той момент, коли всі точки, що залишилися в ньому, мають однакову масу.

12 ***Порядок*** на точках площини визначимо в такий спосіб: $(x, y) \le (i, v)$, якщо або $x<i$, або $x=u$ і $y \le v$. Перелічити точки заданої множини точок на площині відповідно до цього порядку.

13 Задано дві множини точок на площині. Побудувати перетинання й різницю цих множин.

14 Множину точок на площині назвемо ***регулярною***, якщо разом з кожною парою різних точок воно містить також ще одну – третю – вершину правильного трикутника з вершинами в цих точках. Визначити, чи регулярно задана множина точок.

15 На площині задано ***n*** множин по ***m*** точок у кожній. Серед точок першої множини знайти таку, котра належить найбільшій кількості множин.

16 На площині задані множина точок ***A*** і множина кіл ***B***. Знайти дві такі різні точки з ***A***, що пряма, яка проходить через них, перетинається з максимальною кількістю кіл з ***B***.

17 На площині задані множина точок ***A*** и множина прямих ***B***. Знайти дві такі різні точки з ***A***, що пряма, яка проходить через них, паралельна найбільшій кількості прямих з ***B***.

18 На площині задана множина точок ***A*** і точка ***d*** поза нею. Підрахувати кількість (неупорядкованих) різних трійок точок ***a, b*** із ***A*** таких, щоб чотирикутник ***abcd*** був паралелограмом.

19 Визначити радіус і центр кола, що проходить, принаймні, через три різні точки заданої множини точок на площини й містить усередині найбільшу кількість точок цієї множини.

20 Вибрати три різні точки із заданої множини точок на площині так, щоб була мінімальною різниця між кількостями точок, що лежать усередині й поза трикутником з вершинами в обраних точках.

21 Множина попарно різних площин у тривимірному просторі задана перерахуванням трійок точок, через які проходить кожна із площин. Вибрати максимальну підмножину попарно непаралельних площин.

22 Задано множину точок у тривимірному просторі. Знайти мінімум радіусів куль із центрами в цих точках, що містять рівно ***n*** точок цієї множини.

23 Вибрати три різні точки заданої на площині множини точок, що складають трикутник найбільшого периметра.

24 Із заданої на площині множини точок вибрати такі три точки, що не лежать на одній прямій і які складають трикутник найменшої площі.

25 Дано ***3n*** точок на площині, причому ніякі три з них не лежать на одній прямій. Побудувати множину ***n*** трикутників з вершинами в цих точках так, щоб ніякі два трикутники не перетиналися й не містили один одного.

26 Задано множину точок на площині. Вибрати з них чотири різні точки, які є вершинами квадрата найбільшого периметра.

27 Із заданої множини точок на площині вибрати три різні точки ***A, B, C*** так, щоб усередині трикутника ***ABC*** містилася максимальна кількість точок цієї множини.

28 Із заданої множини точок на площині вибрати дві різні точки так, щоб кола заданого радіуса із центрами в цих точках містили усередині себе однакову кількість заданих точок.

29 На площині задана множина точок ***M*** і коло. Вибрати з ***M*** дві різні точки так, щоб найменшою мірою розрізнялися кількості точок у колі, що лежать по різні боки від прямої, що проходить через ці точки.

30 Дано дві непересічних кінцевих множини точок на площині. Визначити коло, що проходить через ***k*** ***(k*** ≥ ***3***) точок кожної із множин.

31 Дано дві множини точок на площині. З першої множини вибрати три різні точки так, щоб трикутник з вершинами в цих точках містив (строго усередині себе) рівну кількість точок першої й другої множин.

32 Дано дві множини точок на площині. Знайти центр і радіус кола, що проходить через ***k*** ***(k*** ≥ 3***)*** точок першої множини й містить строго усередині себе ***m*** точок другої множини.

33 На площині задана множина попарно різних прямих (коефіцієнтами своїх рівнянь). Указати серед них ту пряму, що має максимальне число перетинань із іншими прямими.

34 Дано дві множини точок на площині. Указати центр і радіус кола, що проходить через ***k*** ***(k*** ≥ ***3)*** точок першої множини й містить строго усередині себе рівне число точок першої й другої множини.

35 Дано дві множини точок на площині. Вибрати три різні точки першої множини так, щоб трикутник з вершинами в цих точках покривав всі точки другої множини й мав мінімальну площу.

36 Дано дві множини точок на площині. Вибрати чотири різні точки першої множини так, щоб квадрат з вершинами в цих точках покривав всі точки другої множини й мав мінімальну площу.

37 Дано дві множини точок на площині. Вибрати три різні точки першої множини так, щоб круг, обмежений колом, що проходить через ці три точки, містив всі точки другої множини й мав мінімальну площу.

38 Знайти ромб найбільшої площі з вершинами в заданій множині точок на площині.

39 Підрахувати кількість рівносторонніх трикутників з різними довжинами основ і вершинами в заданій множині точок на площині.

40 Задано множину точок на площині, що не лежать на одній прямій. Визначити мінімальну підмножину точок, після видалення яких залишаються точки, що лежать на одній прямій.

41 Побудувати множину всіх різних опуклих чотирикутників з вершинами в заданій множині точок на площині.

42 Побудувати множину всіх різних гострокутних трикутників з вершинами в заданій множині точок на площині.

43 На площині задана множина точок і коло радіусом ***R*** із центром на початку координат. Побудувати множину всіх трикутників з вершинами в заданих точках, що мають непусте перетинання з окружністю.

44 Із заданої на площині множини точок вибрати три різні точки так, щоб різниця між площею кругу обмеженого колом, що проходить через ці три точки, і площею трикутника з вершинами в цих точках була мінімальною.

45 Серед трикутників з вершинами в заданій множині точок на площині вказати такий, сторони якого містять максимальне число точок заданої множини.

46 Побудувати два трикутники з вершинами в заданій множині точок на площині так, щоб перший трикутник лежав строго усередині другого.

47 На площині задана множина кіл. Два кола ***A*** і ***B*** назвемо ***зв'язаними***, якщо вони перетинаються або існує третє коло ***C*** заданої множини, пов'язана з ***A*** і ***B***. Вибрати максимальну підмножину попарно не зв'язаних одне з одним кіл.

48 Задано множину точок на площині. Перелічити всі різні максимальні підмножини точок, що лежать на одній прямій, які містять більше двох точок.

49 Визначити радіус і центр кола мінімального радіуса, що проходить хоча б через три різні точки заданої множини точок на площині.

50 Знайти таку точку заданої на площині множини точок, сума відстаней від якої до інших мінімальна.

26.2 Матриці, вектори

51 Знайти максимальне із чисел, що зустрічаються в заданій матриці більше одного разу.

52 Відстань між ***k-м*** та ***i-м*** рядками матриці $A = \|a_{ij}\|$ визначається як $\sum_{j=1}^{n} |a_{kj}| \times |a_{lj}|$. Указати номер рядка, максимально віддаленого від першого рядка заданої матриці.

53 Для заданої перестановки ***A*** чисел 1,…,100 знайти таке k≥ 3, при якому $R(k) = \min_{1 \le i \le 100} |A^k(i) - i|$ максимально.

54 Для заданої перестановки A чисел 1,…,100 знайти максимальне k≥ 1, при якому $A^k(i) = i$ для всіх 1≤ i≤ 100.

55 Задано перестановку ***A*** чисел 1,…,100... Для кожного ***i,*** 1≤ ***i***≤ 100, указати таке (залежне від ***i***) мінімальне значення ***k,*** при якому $A^k(i) = i$.

56 Визначити норму заданої матриці $A = \|a_{ij}\|$, тобто число $\max_i \left(\sum_i |a_{ij}|\right)$.

57 Починаючи із центра, обійти по спіралі всі елементи квадратної матриці розміром 13×13 (роздруковуючи їх у порядку обходу).

58 ***Ланцюгом*** вектора ***A = A(1)A(2) .. .A(n)*** називається будь-яка послідовність індексів $i_1,\ldots,i_k$, така, що $A(i_j) = i_{j+1}$ ***(j=1,2, …,k-1)***. Побудувати максимальний за довжиною ланцюг заданого вектора.

59 За заданою квадратною матрицею розміром 10x10 побудувати вектор довжиною 19, елементи якого – максимуми елементів, діагоналей, паралельних головної діагоналі.

60 Два рядки матриці назвемо ***схожими,*** якщо збігаються множини чисел, що зустрічаються в цих рядках. Знайти кількість рядків у максимальній множині попарно несхожих рядків заданої матриці.

61 ***Характеристикою рядка*** цілочисельної матриці назвемо суму її позитивних парних елементів. Переставляючи рядки заданої матриці, розташувати їх відповідно до зростання характеристик.

62 Побудувати цілочисельну матрицю $A = \|a_{ij}\|$ розміром 10x10 у такий спосіб:

$$a_{ij} = \begin{cases} C_i^j \text{ при } & i \geq j, \\ C_j^i \text{ при } & j > i, \end{cases}$$

де C_i^j – число сполучень із ***i*** елементів по ***j***.

63 Знайти номер рядка заданої цілочисельної матриці розміром 10x10, у якому перебуває найдовша серія.

64 Для заданої цілочисельної матриці знайти максимум серед сум елементів діагоналей, паралельних головній діагоналі матриці.

65 Для заданої цілочисельної матриці знайти мінімум серед сум модулів елементів діагоналей, паралельних побічній діагоналі матриці.

66 Дано матрицю ***A*** розміром 20x20. Вважаючи її складеною зі 100 квадратів розміром 2x2 і переставляючи ці квадрати, перетворити ***A*** так, щоб у результуючій матриці для будь-яких двох квадратів ***B*** і ***C*** виконувалася така умова: якщо сума елементів ***B*** менше суми елементів ***C,*** то ***B*** лежить або вище, або лівіше (коли ***B*** і ***C*** на одній горизонталі) квадрата ***C.***

67 Кажуть, що матриця має ***сідлову точку*** a_{ij}, якщо a_{ij} є мінімальним в і-***му*** рядку й максимальним в j-***му*** стовпці. Знайти номер рядка й стовпця якої-небудь сідлової точки заданої матриці.

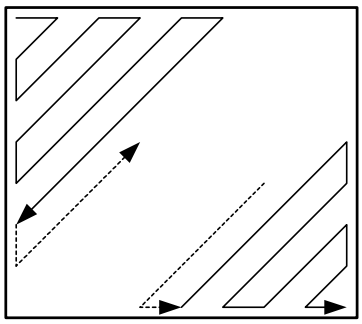
68 Знайти максимальний серед всіх елементів тих рядків заданої матриці, які впорядковані (або за зростанням, або за убуванням).

69 ***Характеристикою стовпця*** цілочисельної матриці назвемо суму модулів її негативних непарних елементів. Переставляючи стовпці заданої матриці, розташувати їх відповідно до зростання характеристик.

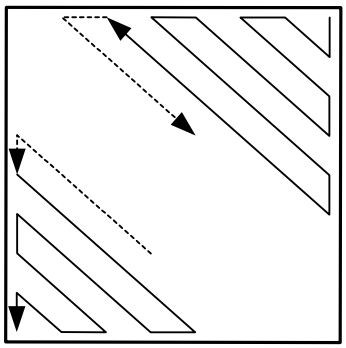
70 Підрахувати кількість стовпців заданої цілочисельної матриці розміром 20x20, які складені з попарно різних чисел.

71 Підрахувати кількість рядків заданої цілочисельної матриці розміром 20x20, що є перестановкою чисел 1, 2, ..., 20.

72 Надрукувати елементи заданої матриці розміром 10х10 у такому порядку, як на рисунку.



73 Надрукувати елементи заданої матриці розміром 10х10 у такому порядку, як на рисунку.



74 Серед рядків заданої цілочисельної матриці, що містять тільки непарні елементи, знайти рядок з максимальною сумою модулів елементів.

75 Серед стовпців заданої цілочисельної матриці, що містять тільки такі елементи, які за модулем не більше 10, знайти стовпець із мінімальним добутком елементів.

76 Для заданої матриці розміром 10х10 знайти такі ***k,*** що ***k***-й рядок матриці збігається з ***k***-м стовпцем.

77 Нехай ***m(A,i)*** означає номер стовпця матриці ***A,*** у якому перебуває останній у рядку мінімум ***i***-го рядка. Перевірити, чи є правильним, що для заданої матриці ***A*** розміром 20х20 виконуються нерівності

m(A, 1)?m(A, 2)?…?m (A, 20).

78 Сусідами елемента a_{ij} у матриці назвемо елементи a_{kj} з $i - 1 \le k \le i + 1$, $j-1 \le l \le j+1$ $(k,l)\neq(i,j)$. Операція

згладжування матриці дає нову матрицю того самого розміру, кожний елемент якої утворюється як середнє арифметичне наявних сусідів відповідного елемента вихідної матриці. Побудувати результат згладжування заданої речовинної матриці розміром 10х10.

79 Будь-який елемент a_{ij} заданої квадратної матриці $A=\|a_{ij}\|$ розміром 20х20 задає розбивку матриці на чотири клітки з індексами ***{(k,l): 1 ≤ до ≤ i, 1 ≤ l≤ i}, {(k, i): 1≤ до<i, j < l ≤ 20}, {(до, l): i < k ≤ 20, i ≤ l</j} і {(k, l): i < k ≤ 20, j < i≤ 20},*** із яких хоча б одна непорожня. Побудувати матрицю $B=\|b_{ij}\|$ того самого розміру, у якій b_{ij} дорівнює мінімуму серед максимальних елементів непустих кліток, обумовлених в A елементом a_{ij}.

80 Визначити, чи є лінійно незалежними три заданих вектори цілих чисел довжиною 30.

81 Перевірити, чи задовольняє задана матриця $A=\|a_{ij}\|$ розміром 10х10 таку умову: для всіх $i>1$ і для всіх $j>1$ правильною є нерівність $a_{ij} \geq a_{i-1,l}+a_{i,l-1}$.

82 Для двох заданих матриць однакового розміру перевірити, чи можна одержати другу матрицю з першої застосуванням (кінцевого числа раз) операцій транспонування щодо головної й побічної діагоналей.

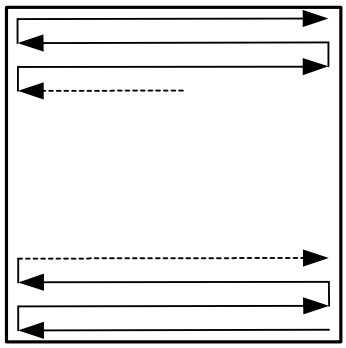
83 Знайти й роздрукувати рядок заданої цілочисельної матриці розміром 10х10, у якому довжина максимальної серії мінімальна.

84 Елемент матриці називається ***локальним мінімумом***, якщо він строго менше всіх наявних у нього сусідів (див. задачу 80). Підрахувати кількість локальних мінімумів заданої матриці розміром 10х13.

85 В умовах попередньої задачі знайти максимум серед всіх локальних мінімумів заданої матриці розміром 10х12.

86 Визначити, чи стає симетричною (щодо головної діагоналі) задана матриця розміром 10х10 після заміни на число 0 кожного локального мінімуму (див. задачу 84).

87 Надрукувати елементи заданої матриці розміром 10х10 у такому порядку, як на рисунку.



88 Взаємно однозначне відображення елементів матриці на себе можна задати за допомогою двох цілочисельних матриць: у першій вказувати номер рядка, куди переходить даний елемент, а в другій – номер стовпця. Побудувати дві матриці, що задають відображення кожного елемента матриці розміром 10x10 на симетричний йому щодо головної діагоналі.

26.3 Послідовності, тексти, речення й слова

89 Перевірити, чи є в заданому тексті баланс відкриваючих і закриваючих круглих дужок, тобто чи є правильним те, що можна встановити взаємно однозначну відповідність відкриваючих і закриваючих дужок з такими властивостями:

а) відкриваюча дужка завжди передує відповідній закриваючій;

б) перший і останній символи тексту – пари відповідних одна одній дужок.

90 Для пар розташованих поруч символів, що зустрічаються в заданому тексті, указати, скільки разів зустрічається в тексті кожне з таких двобуквених сполучень.

91 Для заданого тексту визначити довжину максимальної серії, що міститься в ньому, символів, відмінних від букв.

92 У заданій послідовності цілих чисел знайти найдовшу підпослідовність, що є арифметичною або геометричною прогресією.

93 За заданою послідовністю цілих чисел ***A*** побудувати послідовність ***B*** таку, що ***B (i)*** – це кількість елементів з ***A,*** що перевершують ***A(i),*** у початковому відрізку ***A*** довжиною ***i - 1***.

94 Знайти максимум довжини таких початкових відрізків заданого слова, які мають вигляд ***vv,*** де ***v*** – симетричне слово.

95 Перелічити всі слова заданого речення, які складаються з тих самих букв, що й перше слово речення.

96 Знайти максимальну за довжиною монотонну (тобто або неубувальну, або незростаючу) підпослідовність заданої послідовності цілих чисел.

97 Для заданої послідовності цілих чисел ***A = A(1)A(2) ... A(n)*** визначимо ***T(i,j)*** як $\sum_{k=i}^{j} A(k)$. Знайти ***i, j*** такі, що ***T(i, j)*** максимально.

98 ***Характеристикою*** слова назвемо довжину максимальної серії, що міститься в ньому. Упорядкувати слова заданого речення відповідно до зростання їхніх характеристик.

99 У заданому реченні знайти пару слів, з яких одне є обертанням іншого.

100 Для кожного зі слів заданого речення вказати, скільки разів воно зустрічається в реченні.

101 Знайти найдовше симетричне слово заданого речення.

102 Відстань між двома словами рівної довжини – це кількість позицій, у яких розрізняються ці слова. У заданому реченні знайти пару найбільш далеко віддалених слів заданої довжини.

103 Відредагувати задане речення, видаляючи з нього слова-серії (див. задачу 91), а також ті слова, які вже зустрічалися в реченні раніше.

104 Із заданого тексту вибрати й надрукувати ті символи, які зустрічаються в ньому рівно один раз (у тім порядку, як вони зустрічаються в тексті).

105 У реченні всі слова починаються з різних букв. Надрукувати (якщо можна) слова речення в такому порядку, щоб остання буква кожного слова збігалася з першою буквою наступного слова.

106 Знайти множину всіх слів, які зустрічаються в кожному із двох заданих речень.

107 Відредагувати задане речення, видаляючи з нього всі слова з непарними номерами й перевертаючи слова з парними номерами.

Приклад: HOW DO YOU DO → OD OD.

108 Задано два тексти; кожний текст складений з попарно різних слів. Визначити, чи можна одержати другий текст із першого видаленням деяких його символів.

109 Знайти найдовше спільне слово двох заданих речень.

110 Дано два речення. Знайти найкоротше зі слів першого речення, якого немає в другому реченні.

111 Серед слів заданого речення, які не є серіями, знайти таке, котре має найбільше число входжень у речення.

112 Перевірити, чи є правильним те, що в заданому реченні будь-яке несиметричне слово має парну довжину.

113 Відредагувати задане речення, видаляючи з нього слова, які зустрічаються в реченні задане число раз.

114 Визначити, чи міститься в заданій послідовності цілих чисел хоча б одне число Фібоначчі.

115 Знайти найменше загальне кратне всіх чисел, що містяться в заданій послідовності натуральних чисел.

116 Перевірити, чи є задана послідовність цілих чисел перестановкою початкового відрізка послідовності натуральних чисел.

117 Надрукувати задане речення таким чином, щоб кожне його слово цілком перебувало в одному й тому самому рядку роздруківки (тобто позбутися переносів).

118 Указати довжину такого початкового відрізка заданої послідовності цілих чисел, для якого відношення ступенів двійки, що зустрічаються в ньому, й чисел Фібоначчі максимальне.

119 У заданому реченні вказати слово, у якому частка голосних (A, E, I, O, U*)* максимальна.

120 Переставити й роздрукувати слова заданого речення відповідно до збільшення частки приголосних (B, C, D, F, G, H, K, L, M, N, P, Q, R, S, T, V, W, X, Y, Z) у цих словах.

121 Для кожного символу заданого тексту вказати, скільки разів він зустрічається в тексті. Повідомлення про один символ повинне друкуватися не більше одного разу.

122 Замінити закінчення ING кожного слова, що зустрічається в заданому реченні, на ED.

123 Відредагувати задане речення, видаляючи з нього всі слова, цілком складені із входжень не більш ніж двох букв.

Приклад: AKKA KNOBIKAISE > KNOBIKAISE.

124 Задано текст, у якому немає входжень символів '(' і ')'. Виконати його ***стиснення,*** тобто замінити будь-яку максимальну підпослідовність, складену з більш ніж трьох входжень того самого символу, на ***(k)s,*** де ***s*** – повторюваний символ, а ***k > 3*** – кількість його повторень.

125 Відредагувати задане речення, заміняючи будь-яке входження слова вигляду $\alpha\beta\overline{\alpha}$ на ***α*** де ***α, β*** – підслова, а $\overline{\alpha}$ – обертання слова ***α***.

126 Задано речення, у якому є входження кожної з букв латинського алфавіту. Нехай C_a для букви ***α*** означає перше зі слів речення, що містять ***α***. Перевірити, чи є правильним, що довжини слів C_a упорядковані відповідно до порядку букв алфавіту (тобто слово C_A не довше слова C_B і т. д.).

127 Побудувати послідовність довжиною 100, утворену цифрами п’ятіркового подання послідовності натуральних чисел, що починається із заданого ***n.***

26.4 Задачі із цілими числами

128 Знайти такі всі прості числа, що не більше заданого ***N***, двійковий запис яких являє собою симетричну послідовність нулів і одиниць (що починається з одиниці!).

129 Побудувати таблицю всіх різних розбиттів заданого цілого числа ***N > 0*** на суму трьох натуральних складників (розбиття, що відрізняються лише порядком складників, різними не вважаються).

130 Знайти всі прості числа, не більші заданого ***N>0***.

131 Для заданого натурального ***N*** визначити найменше число ***S***, яке можна подати у вигляді суми ***a+b*** принаймні двома різними способами (***a, b*** – натуральні числа; подання, що відрізняються лише порядком складників, різними не вважаються).

132 Перетворити задане ціле число з ***p***-кової системи числення в ***q***-кову ***(p, q ≤ 16***; вихідне число має не більше ***n*** знаків).

133 Розкласти задане натуральне число на прості множники.

134 Побудувати перші ***N*** натуральних чисел, дільниками яких є тільки числа 2, 3 і 5.

135 Указати те число заданої множини цілих чисел, у двійковому поданні якого найбільше одиниць.

136 Перелічити всі натуральні числа, що не більше заданого ***N***, у двійковому поданні яких номери ненульових розрядів утворять арифметичну прогресію.

137 Серед простих чисел, що не більше ***N,*** знайти таке, у двійковому записі якого максимальне число одиниць.

138 Перелічити все пари «сусідніх» простих чисел, що не більші ***N,*** трійкові подання яких утворюються один із одним записом цифр у зворотному порядку (перша така пара – це 5 і 7).

139 Знайти всі натуральні числа, що не більші заданого ***N*** і дорівнюють сумі кубів своїх цифр.

140 Знайти усі пари двозначних натуральних чисел ***M, N*** таких, що значення добутку ***M*N*** не зміниться, якщо поміняти місцями цифри кожного зі співмножників (такою парою буде, наприклад, 38 і 83).

141 Знайти всі натуральні числа, не більші заданого ***N,*** такі що діляться на кожну зі своїх цифр.

142 Задано три натуральних числа ***A, B*** і ***N***. Знайти всі натуральні числа, не більші ***N***, які можна подати у вигляді суми (довільного числа) доданків, кожне з яких – ***A*** або ***B.***

143 Знайти всі натуральні числа, не більші заданого ***N***, десятковий запис яких є строго зростаючим або строго убувальною послідовністю цифр.

144 Визначити, чи можна подати задане натуральне число як суму кубів яких-небудь трьох натуральних чисел.

145 Визначити, чи є періодичною послідовністю двійковий запис заданого натурального числа ***N,*** тобто чи має вона вигляд aa...a, де a – деяка непуста послідовність.

146 Знайти всі натуральні числа, не більші заданого *N*, що можна подати у вигляді суми квадратів двох яких-небудь різних натуральних чисел.

147 Задано множину натуральних чисел. Замінити кожне з них на число, що виходить із вихідного записом його (десяткових) цифр у зворотному порядку. Отриману множину чисел роздрукувати.

148 Для натуральних ***A, B*** операцію $\oplus$ визначимо так: $A \oplus B = A - B + A \ MOD \ B$.

Знайти всі такі пари ***A, B,*** не більші заданого ***N,*** для яких $A \oplus B = B \oplus A$.

26.5 Рекурсія

149 Написати рекурсивну функцію для знаходження біноміальних коефіцієнтів, користуючись їхнім визначенням.

$$C_n^m = \begin{cases} 1, & \text{если } m = 0, n > 0 \text{ или } m = n \geq 0, \\ 0, & \text{если } m > n \geq 0, \\ C_{n-1}^{m-1} + C_{n-1}^m & \text{в остальных случаях.} \end{cases}$$

Обчислити C_{2k}^k для k = 2, 4, 6, 8.

150 Задано кінцеву множину імен жителів якогось міста, причому для кожного з жителів перераховані імена його дітей. Жителі ***X*** і ***Y*** називаються ***родичами,*** якщо

а) або ***X*** – дитина ***Y,***

б) ***Y*** — дитина ***X,***

в) існує якийсь ***Z*** такий, що ***X*** є родичем ***Z,*** а ***Z*** є родичем ***Y***. Перелічити усі пари жителів міста, які є родичами.

151 Підрахувати кількість різних зображень заданого натурального ***n*** у вигляді суми не менше двох попарно різних додатних доданків. Подання, що відрізняються лише порядком розташування, різними не вважаються.

152 Надрукувати поле для гри в шахи, у якому чорні поля зображуються квадратами, заповненими 5x5 = 25 зірочками (символами ‘*’).

153 Обчислити визначник заданої матриці, користуючись формулою розкладання по першому рядку: $\det A = \sum_k (-1)^{k+1} a_{1k} * \det(B_k),$ де матриця B_k утворюється із ***A*** викреслюванням першого рядка й ***k***-го стовпця.

154 Напишіть дві функції обчислення ***i***-го числа Фібоначчі – рекурсивну і нерекурсивну – і надрукуйте таблицю для

порівняння часів обчислення ***i***-го числа Фібоначчі для ***i*** = 4, 6, 8, 10, 12 за допомогою цих функцій.

155 Функція ***f(n)*** визначена для цілих додатних чисел у такий спосіб:

$$f(n)=\begin{cases}1, & если \quad n=1,\\ \sum\limits_{i=2}^{n} f(n \quad DIV \quad i), & если \quad n\ge 2.\end{cases}$$

Обчислити ***f(k)*** для ***k = 15, 16, … , 30***...

156 Побудувати синтаксичний аналізатор для поняття ***простое-выражение***.

$$простое \quad выражение \quad ::=\left\{\begin{array}{c} простой-идентификатор \\ (простое-выражение \quad знак-операции \quad простое-выражение)\end{array}\right\}$$

$$знак-операции ::= \left\{\begin{array}{c}-\\+\\*\end{array}\right\}.$$

157 Побудувати синтаксичний аналізатор для поняття ***идентификатор***.

$$èäåíòèôèêà \quad òîð ::= \left\{\begin{array}{c} áóêâà \\ èäåíòèôèêà \quad òîð \quad \left\{\begin{array}{c}öèôðà\\áóêâà\end{array}\right\}\end{array}\right\}.$$

158 Побудувати синтаксичний аналізатор для поняття ***вещественное-число***.

$$вещественное\_число ::= \left\{\begin{array}{c} целое-число, целое-без-знака \\ целое-число, целое-без-знака \quad E \quad целое-число \\ целое-число \quad E \quad целое-число\end{array}\right\},$$

$$целое-без-знака ::= \quad цифра \quad \{цифра\},$$

$$целое-число ::= \left\{\begin{array}{c} целое-без-знака \\ \left\{\begin{array}{c}+\\-\end{array}\right\} \quad целое-без-знака\end{array}\right\}.$$

159 Побудувати синтаксичний аналізатор для поняття ***простое-логическое***.

$$простое-логическое ::= \left\{\begin{array}{c} TRUE \\ FALSE \\ простой-идентификатор \\ NOT \quad простое-логическое \\ (простое-логическое \quad знак-операции \quad простое-логическое)\end{array}\right\}$$

$$простой-идентификатор \quad ::= \quad буква,$$

$$знак-операции \quad ::= \quad \left\{ \begin{matrix} AND \\ OR \end{matrix} \right\}.$$

160 Написати програму, що за заданим ***простым логическим*** виразом (визначення поняття міститься у формулюванні попередньої задачі), що не містить входжень простих ідентифікаторів, обчислює й друкує значення цього виразу.

161 Побудувати синтаксичний аналізатор для поняття ***константное-выражение***.

162 Написати програму, що за заданим ***константным-выражением*** (визначення поняття міститься у формулюванні попередньої задачі) обчислює й друкує або значення цього виразу, або повідомлення «при обчисленні константного виразу отримано проміжний результат, що перевершує за модулем мільйон».