МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”

**Н. Б. Яворський, У. Б. Марікуца, М. І. Андрійчук, І. В. Фармага**

**ЛАБОРАТОРНИЙ ПРАКТИКУМ**
**З ДИСЦИПЛІНИ**

# АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Навчальний посібник

*Рекомендувала Науково-методична рада*
*інституту комп’ютерних наук та інформаційних технологій*
*Національного університету “Львівська політехніка”*

Львів
Видавництво Львівської політехніки
2018

УДК XXX.XXX
ББК XX.XX

**Рецензенти:** **Миклушка І. З.**, к.т.н., декан факультету видавничо-поліграфічної, інформаційної технології Української академія друкарства, доцент;
**Малець І. О.**, к. т. н., професор кафедри управління проектами, інформаційних технологій та телекомунікацій Львівського державного університету безпеки життєдіяльності, полковник служби цивільного захисту, доцент;
**Шпак З. Я.**, к.т.н., доцент кафедри автоматизованих систем управління інституту комп'ютерних наук та інформаційних технологій Національного університету "Львівська політехніка", доцент.

*Рекомендувала Науково-методична рада інституту комп'ютерних наук та інформаційних технологій Національного університету "Львівська політехніка", як навчальний посібник для студентів напряму 122–"Комп'ютерні науки", 126–"Інформаційні системи та технології", 015– "Професійна освіта (комп'ютерні технології)"*
*(протокол № ___ від _____2018 р.).*

**Яворський Н. Б.**

Лабораторний практикум з дисципліни "Алгоритмізація та програмування": навчальний посібник / Н. Б. Яворський, У. Б. Марікуца, М. І. Андрійчук, І. В. Фармага – Львів : Видавництво Львівської політехніки, 2018. – 191 с.

ISBN XXX-XXX-XXX-XXX-X

Метою посібника є полегшення студентам засвоєння матеріалу дисципліни алгоритмізація та програмування. Посібник написаний таким чином, що для освоєння матеріалу достатньо шкільних знань. Складні теми супроводжуються детальними поясненнями та ілюстративними прикладами.

Розкрито основні аспекти алгоритмізації, наведено деякі поширені алгоритми та їх графічне представлення відповідно до чинних стандартів. Основні аспекти програмування розкриті на прикладі мови програмування С (Сі) у восьми лабораторних роботах.

Для студентів молодших курсів комп'ютерних спеціальностей та тих, хто хоче розпочати вичення програмування.

**УДК XXX.XXX**
**ББК XX.XX**



ISBN XXX-XXX-XXX-XXX-X

# ЗМІСТ

# ВСТУП

Навчальний посібник “Лабораторний практикум з дисципліни алгоритмізація та програмування” базується на курсі лекцій та інструкцій до виконання лабораторних робіт за відповідною дисципліною, що викладаєься на кафедрі система автоматизованого проектування національного університету “Львівська політехніка”. Він написаний для студентів молодших курсів комп’ютерних спеціальностей, однак простота і послідовність викладення матеріалу робить його загальнодоступним. Посібник може використовуватися для самостійного вивчення дисципліни.

Метою посібника є полегшення студентам засвоєння матеріалу дисципліни алгоритмізація та програмування. Посібник написаний таким чином, що для освоєння матеріалу достатньо шкільних знань. Складні теми супроводжуються детальними поясненнями та ілюстративними прикладами. Теоретичні відомості наведені в посібнику базуються на роботах таких корифеїв алгоритмізації та програмування, як *Thomas Cormen* (американський вчений, професор інформатики та комп'ютерних наук), *Niklaus Wirth* (швейцарський вчений, професор інформатики та комп'ютерних наук), *Stephen Prata* (американський вчений, професор фізики та астрономії, спеціаліст в області програмування та дискретної математики) та інших.

Посібник відрізняється за структурою від загальноприйнятої форми подачі матеріалу, що викликано методикою викладання. Весь матеріал поділено на окремі теми (лабораторні роботи), які містять теоретичну частину і окремі завдання, за допомогою яких студент засвоюватиме певну тему. У теоретичній частині розкрито основні аспекти алгоритмізації, наведено деякі поширені алгоритми та їх графічне представлення відповідно до чинних стандартів. Наведено основні аспекти програмування на прикладі мови С (*Ci*) за конкретною темою. Наведено багато прикладів, що полегшують засвоєння матеріалу а також індивідуальні завдання до кожної лабораторної роботи, що спрямовані на самостійне опрацювання для закріплення матеріалу.

С це потужна мова програмування для професіоналів, яка однаково популярна як в середовищі любителів, так і в середовищі програмістів, що пишуть програми для комерційного застосування. Заглянемо в історію її створення.

Давним-давно в 1966-у році у Кембриджському університеті *Martin Richards* (британський вчений, програміст) розробив мову програмування *BCPL* (*Basic Combined Programming Language*), що призначалася для створення компіляторів для інших мов програмування. BCPL відрізнялася від інших тим, що у ній були відсутні функції, які ускладнювали компіляцію і таким чином, програми могли легко переноситися на різні платформи. У той час це було

велике досягнення! Саме на BCPL у 1967 була вперше написана програма "*Hello world!*".

У 1969 зусиллями *Ken Thompson* (американський вчений, програміст) та при підтримці *Dennis Ritchie* (американський вчений, програміст) в AT&T Bell Labs на світ з'явилася урізана версія BCPL з сильно зміненим, тепер вже звичним синтаксисом під назвою B (*Бі*). У професійних колах BCPL та B жартома розшифровують як *Before C Programming Language*.

Нарешті, в 1972, під час роботи Dennis Ritchie та Ken Thompson над створенням операційної системи UNIX, на основі мови B, була створена мова програмування C. Вона замислювався як інструментальний засіб для програмістів-практиків. Більшість мов програмування створювалися з метою бути корисними, але досить часто перед ними ставилися інші завдання. Наприклад, спочатку мова Pascal призначалася для полегшення вивчення принципів програмування. Мова BASIC створювалася як мова програмування, наближена до англійської мови, щоб полегшити вивчення програмування студентам, не знайомим з комп'ютерами. Такі підходи не завжди відповідали прагматичному підходу до вирішення повсякденних задач. Тому створення мови C як мови, призначеної для програмістів, зробила її однією з найбільш затребуваних.

C є сучасою мовою програмування, що включає в себе засоби керування, які теорія і практика обчислювальної техніки розглядає як корисні і бажані. Її конструкція добре підходить для низхідного планування, для структурного програмування і для модульного проектування. Все це дає змогу отримувати надійні і зрозумілі програми. З іншої сторони свобода мови C вимагає додаткової відповідальності. Зокрема, використання в C вказівників означає можливість появи програмних помилок, які важко відстежити. Ціною свободи є постійна пильність.

У дев'яностих роках минулого століття багато компаній, що виготовляють і поставляють програмне забезпечення, при реалізації великих програмних проектів стали переходити на мову C++. Мова C++ додає до C інструментальні засоби об'єктно-орієнтованого програмування. (*Об'єктно-орієнтоване програмування* є парадигмою, яка намагається формувати мову таким чином, щоб вона відповідала завданню, на відміну від формулювання завдання так, щоб воно відповідало мові програмування.) У першому наближенні C++ можна розглядати як надмножину мови C в тому сенсі, що програма на C також є, або майже є, програмою на C++. Вивчаючи мову C, ви фактично вивчаєте багато аспектів C++.

Незважаючи на популярність новіших мов на зразок C++, мова C зберігає лідируюче положення по здатності вирішувати завдання з області розроблення програмного забезпечення, зазвичай входячи в десятку найбільш затребуваних

мов програмування. Зокрема, С незамінно використовується для програмування вбудованих систем.

Зараз існує багато реалізацій мови С. В ідеальному випадку, коли ви пишете програму на С, вона повинна працювати однаково на будь-який реалізації за умови, що в ній не використовується код, специфічний для конкретної машини. Щоб домогтися цього на практиці, різні реалізації повинні відповідати загальновизнаним стандартам. Спочатку для мови С не існувало офіційного стандарту. Загальновизнаним стандартом служило перше видання книги *Brian Kernighan* (канадський вчений, програміст) та Dennis Ritchie "*The C Programming Language*". Цей стандарт отримав позначення *K&R C*.

У міру того як мова С розвивалася і отримувала все більш широке застосування в різних системах, спільнота користувачів С відчула гостру потребу у всеосяжному, сучасному і строгому стандарті. Щоб задовольнити цю потребу, інститут *ANSI* (*American National Standards Institute*) утворив в 1983 році спеціальний комітет, метою якого була розробка нового стандарту, і він формально був прийнятий в 1989 році. Цей стандарт (*ANSI C*) визначає як саму мову, так і стандартну бібліотеку С. Організація *ISO* (*International Organization for Standardization*) прийняла стандарт мови С (*ISO C*) в 1990 році. По суті ISO C та ANSI C є одним і тим же стандартом. Остаточну версію ANSI / ISO часто називають *C89* або *C90*. Комітет висунув кілька основних принципів. Найбільш цікавим був – "*зберігати дух мови С*":

- довіряти програмісту;
- не перешкоджати йому робити те, що він вважає за необхідне;
- не збільшувати мову і зберігати її простоту.
- передбачати тільки один спосіб виконання операції.
- робити операцію швидкодійною, навіть коли не гарантується переносимість.

У 1994 році об'єднаний комітет ANSI / ISO, почав роботу з перегляду існуючого стандарту, результатом якої став стандарт *C99*. Комітет підтвердив базові принципи стандарту С90, в тому числі принцип малого розміру та простоти мови С. Мета, озвучена комітетом, полягала в тому, щоб не додавати в мову нові властивості за винятком тих, які необхідні для досягнення нових цілей, поставлених перед мовою розвитком інформаційних технологій (інтернаціоналізація, кодування, розширені 64-розрядні архітектури, застосування в наукових задачах на заміну FORTRAN, тощо).

Підтримка стандарту – процес нескінченний, і в 2007 році комітет приступив до створення наступної, найновішої на сьогодні версії стандарту, яка була випущена як *C11*. Комітет висунув ряд нових принципів, одним з яких стало деяке пом'якшення мети "довіри програмісту" з урахуванням сучасної турботи про захищеність і безпеку програмного коду. Перегляд стандарту був

обумовлений в основному появою нових технологій. Один із прикладів – додавання підтримки паралельного програмування на рівні мови, а не сторонніх бібліотек у відповідь на тенденцію застосування декількох процесорів в комп'ютерах. У даному навчальному посібнику викладення матеріалу стосовно мови С відповідає стандарту С11.

На відміну від інших подібних навчальних посібників, що в основному наводять *MS Visual Studio*, в якості інтегрованого середовища розроблення програмних продуктів, у цьому використовується *Code::Blocks* – вільне кросплатформенне середовище, що активно розвивається. Перевагами цього середовища над MS Visual Studio є робота в операційних системах Linux та macOS. Однак, використання саме цього середовища не є принциповим. Студент вільний обирати собі будь-які інструменти, що вважає зручними.

Вивчення алгоритмізації та програмування немає кінця. З розвитком комп’ютерних технологій з’являються нові задачі та їх нові вирішення. У даному навчальному посібнику розглянуті тільки основні аспекти. Так з розгляду опущено такі аспекти мови С, як бітові поля, розширене представлення даних у вигляді списків та дерев, багатопотоковість та міжпроцесорна взаємодія, масиви змінної довжини; тільки частково розглянуті перечислення, класи зберігання у пам’яті, динамічне виділення пам’яті, вказівники на функції. Їх вивчення виходить за рамки цього посібника, тому відправляємо зацікавленого читача до джерел інформації, наведених в кінці.

Зауваження та пропозиції щодо покращення лабораторного практикуму просимо надсилати на електронну скриньку nazarii.b.yavorskyi@lpnu.ua.

*"...Спочатку вивчіть науку програмування і всю теорію. Далі виробіть свій програмістський стиль. Потім забудьте все і просто програмуйте..."*

George Carrette, 1990, американський програміст.

*"...Завжди пишіть код так, ніби супроводжувати його буде схильний до насильства психопат, який знає, де ви живете. Код існує для зручності читання!..."*

John Woods, 1991, британський програміст,
розробник комп'ютерних ігор.

*KISS – "keep it simple, stupid!" / "не ускладнюй, дурню!"*

принцип проектування систем,
який вперше запропонував у 1960-их роках
Kelly Johnson, американський авіаконструктор.

# ОСНОВИ АЛГОРИТМІЗАЦІЇ ТА ПРОГРАМУВАННЯ

## 1.1. Алгоритми

### 1.1.1.Визначення алгоритму

Що таке *алгоритм*? Узагальнений відповідь [1] – *набір кроків для виконання завдання*. Є алгоритми, які виконуються у повсякденному житті. Наприклад, алгоритм чищення зубів: відкрити тюбик зубної пасти, взяти зубну щітку, видавлювати зубну пасту на щітку до тих пір, поки не буде достатньо для покриття щітки, закрити тюбик, помістити щітку в один з квадрантів рота, переміщати щітку вгору і вниз $N$ секунд і т.д. Якщо доводиться їздити на роботу, звичайно ж, є алгоритм вибору транспорту і місць пересадки і так далі...

Розглянемо алгоритми, що виконуються на комп'ютерах, або, більш узагальнено, на обчислювальних пристроях. Ці алгоритми так само впливають на повсякденне життя, як і алгоритми, які виконує безпосередньо людина. Можна використовувати GPS для пошуку маршруту поїздки – працює алгоритм пошуку найкоротшого шляху. Купуєте щось в Інтернеті? – значить, використовується (або принаймні повинен використовуватися) захищений веб-сайт, на якому працює алгоритм шифрування. Коли робляться покупки в Інтернеті, вони доставляються службою доставки – використовується алгоритм розподілу замовлень по машинам, а потім визначається порядок, в якому кожен водій повинен доставити пакети. Алгоритми працюють на комп'ютерах всюди – на ноутбуці, на серверах, на смартфоні, у вбудованих системах (таких, як автомобіль, мікрохвильова піч або системи клімат-контролю) – скрізь!

У чому ж відмінність алгоритму, який працює на комп'ютері, від алгоритму, який виконує безпосередньо людина? Людина може стерпіти, якщо алгоритм описаний неточно, але комп'ютер не настільки терплячий. Наприклад, якщо людина їдете на роботу, вона може собі сказати,– "якщо дорога перевантажена, вибирай інший маршрут!". Людина можете зрозуміти, що значить “перевантажена дорога”, але комп'ютеру такі тонкощі невідомі.

Таким чином, *комп'ютерний алгоритм – є набором кроків для виконання завдання, описаних досить точно для того, щоб комп'ютер міг їх виконати*. Комп'ютерні алгоритми вирішують обчислювальні завдання. З огляду на це від них потрібні дві речі: вони повинні завжди давати правильне рішення поставленої задачі і ефективно використовувати обчислювальні ресурси.

### 1.1.2.Коректність алгоритму

Що це означає – отримання правильного рішення задачі? Зазвичай можна точно визначити, що спричинить за собою правильне рішення. Наприклад, якщо GPS видає правильне рішення задачі пошуку найкращого маршруту для

подорожі, то це буде маршрут, по якому можна добиратися в бажаний пункт призначення швидше, ніж при поїздці по будь-якому іншому маршруту. Але коли дорога перевантажена, а треба дістатися швидше, GPS може дати погану пораду. Таким чином, алгоритм може бути правильним навіть при невірних вхідних даних. Для деяких завдань досить складно, а то й просто неможливо сказати, чи дає алгоритм вірне рішення задачі. Наприклад, в разі оптичного розпізнавання іноді важко сказати це цифра “ 5 ” чи латинська буква “ S ”. Буває, однак, що можна вважати рішенням алгоритм, який іноді дає некоректні результати, якщо тільки є можливість контролювати, як часто це відбувається. Поняття *коректності* ділиться на два типи:

- *часткова коректність* – програма дає коректний результат для тих випадків, коли вона завершується;
- *повна коректність* – програма завершує роботу та видає коректний результат для всіх елементів з діапазону вхідних даних.

### 1.1.3.Ефективність алгоритму

Що означає, що алгоритм ефективно використовує обчислювальні ресурси? Один з основних показників *ефективності* це *час роботи алгоритму*. Алгоритм, який дає правильне рішення, але вимагає великого часу для його отримання, не має практичної цінності. Якби GPS витрачав годину, щоб визначити маршрут руху, стали б витрачатися сили, щоб його включити? Час є основним показником ефективності, який використовується для оцінки алгоритму, звичайно, після того, як буде показано, що алгоритм дає правильне рішення. Але це не єдина міра ефективності. Слід також враховувати, яка кількість пам'яті комп'ютера потрібна алгоритму для роботи, так як інакше може виявитися, що він просто не стане працювати через нестачу пам'яті. Інші можливі ресурси, які може використовувати алгоритм, це мережеві з'єднання, дискові операції, тощо.

Так як же все-таки можна оцінювати швидкість алгоритму? Відповідь полягає в тому, щоб робити це за допомогою об'єднання двох ідей. Поперше, визначається залежність часу роботи алгоритму від розміру його вхідних даних. Наприклад, при пошуку заданого елемента в списку, щоб визначити, чи є він там чи ні, розміром вхідних даних є кількість елементів у списку.

По-друге, визначається як швидко з ростом розміру вхідних даних зростає час роботи алгоритму – *швидкість росту* часу роботи. При цьому, значущим є тільки домінуючий член функції часу роботи алгоритму, а інші члени і можливі коефіцієнти не враховуються. Тобто, значущим є тільки порядок зростання часу роботи. Час, який витрачає алгоритм як функція від розміру задачі $n$, називають *часовою складністю* цього алгоритму $T(n)$. Асимптотику поведінки цієї функції при збільшенні розміру задачі називають *асимптотичною часовою складністю*,

а для її позначення використовують нотацію Ландау (велике $O$) [1], [2]. Саме асимптотична складність визначає розмір задач, які алгоритм здатен обробити. Наприклад, якщо алгоритм обробляє вхідні дані розміром $n$ за час $cn^2$, де $c$ – деяка стала, то кажуть, що часова складність такого алгоритму $O(n^2)$. Приклади:

$O(1)$ Сталий час роботи не залежно від розміру задачі. Наприклад, прочитати деякий елемент масиву.

$O(\log \log n)$ Дуже повільне зростання необхідного часу. Наприклад, коли задача розбивається на незалежні підзадачі пропорційно до квадратного кореня від розміру вхідних даних.

$O(\log n)$ Логарифмічне зростання – подвоєння розміру задачі збільшує час роботи на сталу величину. Наприклад, коли задача розбивається на незалежні підзадачі діленням їх навпіл.

$O(n)$ Лінійне зростання – подвоєння розміру задачі подвоїть і необхідний час. Наприклад прочитати деякий елемент списку, або знайти необхідний елемент в масиві.

$O(n \log n)$ Лінеаритмічне зростання – подвоєння розміру задачі збільшить необхідний час трохи більше ніж вдвічі. Наприклад, коли задача розбивається на підзадачі поділом навпіл, а потім окремі результати збираються в єдиний.

$O(n^2)$ Квадратичне зростання – подвоєння розміру задачі вчетверо збільшує необхідний час. Наприклад для простих алгоритмів сортування, де зустрічається вкладений цикл по всіх вхідних даних.

$O(n^3)$ Кубічне зростання – подвоєння розміру задачі збільшує необхідний час у вісім разів. Наприклад, просте множення матриць, або алгоритми де зустрічається потрійне вкладення циклів по всіх елементах.

$O(c^n)$ Експоненціальне зростання – збільшення розміру задачі на 1 призводить до $c$-кратного збільшення необхідного часу; подвоєння розміру задачі підносить необхідний час у квадрат. Наприклад, алгоритми повного перебору.

### 1.1.4. Властивості алгоритму

Алгоритми мають ряд важливих властивостей:

- *Скінченність* – алгоритм має завжди завершуватись після виконання скінченної кількості кроків.
- *Дискретність* – процес, що визначається алгоритмом, можна розділити на окремі кроки.
- *Визначеність* – кожен крок алгоритму має бути точно визначений. Дії, які необхідно здійснити, повинні бути чітко та недвозначно визначені для кожного можливого випадку.
- *Вхідні дані* – алгоритм має деяку множину вхідних даних.
- *Вихідні дані* – алгоритм має множину вихідних даних.
- *Масовість* – алгоритм повинен забезпечувати розв'язання будь-якої задачі з класу однотипних задач за будь-якими вхідними даними, що належать до області застосування алгоритму.

### 1.1.5. Представлення алгоритму

У процесі розробки алгоритму можуть використовуватись різні способи його опису, які відрізняються за простотою, наочністю, компактністю, мірою формалізації, орієнтації на машинну реалізацію тощо. Форми запису алгоритму:

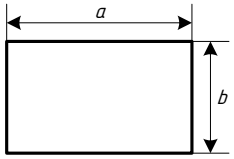
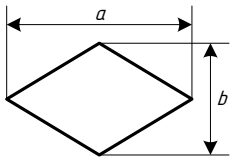
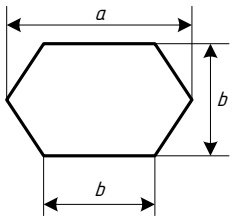
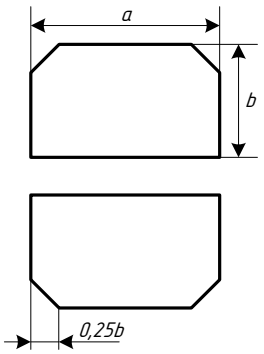
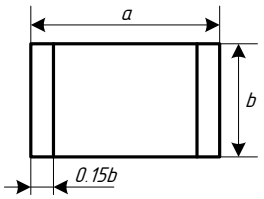
- словесна або вербальна (мовна, формульно-словесна);
- псевдокод (формальні алгоритмічні мови);
- схемна (структурограми, блок-схеми).

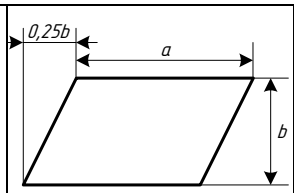
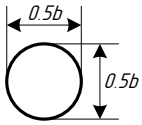
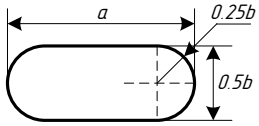
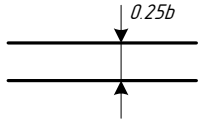
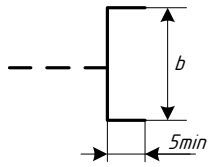
## 1.2. Графічне представлення алгоритмів

Одним з можливих графічних описів алгоритму є використання стандартизованих [3] блок-схем[1]. Схеми алгоритмів складаються з символів, що мають задане значення короткого пояснювального тексту і з'єднувальних ліній.

При зображенні символів рекомендується дотримуватися строгих розмірів, визначених двома значеннями $a$ і $b$. Значення $a$ вибирається з ряду 15, 20, 25, ... мм, $b$ розраховується зі співвідношення $2a = 3b$. З введенням стандарту ГОСТ 19.701-90 / ISO 5807-85 [3] визначення розмірів несе тільки рекомендаційний характер, але, при їх дотриманні блок-схеми мають більш акуратний вигляд:

[1] Іншим поширеним способом є використання UML (*Unified Modeling Language*) діаграм, зокрема діагами діяльності. Однак, UML слід розглядати значно ширше, ніж інструмент для опису алгоримів – це уніфікована мова моделювання, що використовується у парадигмі об'єктно-орієнтованого програмування і є невід'ємною частиною уніфікованого процесу розробки програмного забезпечення. UML використовує графічні позначення для визначення, візуалізації, проектування й документування цілісних програмних систем.

| | | |
|---|---|---|
| Операція (процес) |  | Символ відображає функцію обробки даних будь-якого виду (виконання певної операції або групи операцій, що приводить до зміни значення, форми або розміщення інформації або до визначення, за яким з декількох напрямків потоку слід рухатися). |
| Розв'язання (умова) |  | Символ відображає рішення або функцію перемикача типу, що має один вхід і ряд альтернативних виходів, один і тільки один з яких може бути активований після обчислення умов, визначених всередині цього символу. Результати обчислення можуть бути записані по сусідству з лініями, що відображають ці шляхи. |
| Підготовчий процес |  | Символ відображає модифікацію команди або групи команд з метою впливу на деяку подальшу функцію (установка перемикача, модифікація індексного регістра, ініціалізація програми, модифікація лічильника циклу). |
| Межі циклу |  | Символ, що складається з двох частин, відображає початок і кінець циклу. Обидві частини символу мають один і той же ідентифікатор. Умови для ініціалізації, збільшення, завершення і т.д. поміщаються всередині символу на початку або в кінці в залежності від розташування операції, що перевіряє умову. |
| Процедурний процес |  | Символ відображає зумовлений процес, що складається з однієї або декількох операцій або кроків програми, які визначені в іншому місці (в підпрограмі, модулі). |

| | | |
|---|---|---|
| Дані |  | Символ відображає дані, носій даних не визначено. Зазвичай позначає введення або виведення даних. |
| З’єднувач |  | Символ відображає вихід в частину схеми і вхід з іншої частини цієї схеми та використовується для обриву лінії і продовження її в іншому місці. Відповідні символи-з'єднувачі повинні містити одне і те ж позначення. |
| Лінія |  | Символ відображає потік даних або управління. При необхідності або для підвищення зручності читання можуть бути додані стрілки-вказівники. |
| Термінатор |  | Символ відображає вихід у зовнішнє середовище і вхід із зовнішнього середовища (початок або кінець схеми програми, зовнішнє використання і джерело або пункт призначення даних). |
| Пунктирна лінія |  | Символ відображає альтернативний зв'язок між двома або більше символами. Крім того, символ використовують для обведення анотованої ділянки. |
| Паралельна дія |  | Символ відображає синхронізацію двох або більше паралельних операцій. |
| Коментар |  | Символ використовують для додавання описових коментарів або пояснювальних записів з метою пояснення або приміток. Пунктирні лінії в символі коментаря пов'язані з відповідним символом або можуть обводити групу символів. Текст коментарів або приміток повинен бути поміщений близько обмежувальної фігури. |

| | | |
|---|---|---|
| Пропуск | ___ . . . ___ | Символ (еліпсис) використовують в схемах для відображення пропуску символу або групи символів, в яких не визначені ні тип, ні число символів. Символ використовують тільки в символах лінії або між ними. Він застосовується головним чином в схемах, що зображують спільні рішення з невідомим числом повторень. |

1.2.1.Правила застосування символів

Символ призначений для графічної ідентифікації функції, яку він відображає, незалежно від тексту всередині цього символу.

Символи в схемі повинні бути розташовані рівномірно. Слід дотримуватися розумної довжини з'єднань і мінімального числа довгих ліній. Більшість символів задумано так, щоб дати можливість розміщення тексту всередині символу. Не повинні змінюватися кути і інші параметри, що впливають на відповідну форму символів. Символи повинні бути, по можливості, одного розміру. Вони можуть бути накреслені в будь-якій орієнтації, але, кращою є горизонтальна.

Мінімальна кількість тексту, необхідного для розуміння функції конкретного символу, слід поміщати всередині цього символу. Текст для читання повинен записуватися зліва направо і зверху вниз незалежно від напрямку потоку. Якщо обсяг тексту, який вміщується всередині символу, перевищує його розміри, слід використовувати символ коментаря.

У схемах може використовуватися ідентифікатор символів. Пов'язаний з даним символом, він декларує його для використання в довідкових цілях у інших елементах документації (наприклад, в лістингу програми). Ідентифікатор символу повинен розташовуватися зліва над символом. Координати зони символу або порядковий номер проставляють у верхній частині символу в розриві його контуру. У схемах може використовуватися опис символів – будь-яка інша інформація, наприклад, для відображення особливого застосування символу з перехресним посиланням, або для поліпшення розуміння функції як частини схеми. Опис символу має бути розташований праворуч над символом.

1.2.2.Правила застосування з’єднань

Потоки даних або потоки управління в схемах показуються лініями. Напрямок потоку зліва направо і зверху вниз вважається стандартним. У випадках, коли необхідно внести більшу ясність в схему (наприклад, при

з'єднаннях), на лініях використовуються стрілки. Якщо потік має напрямок, відмінний від стандартного, стрілки повинні вказувати цей напрямок.

У схемах слід уникати перетину ліній. Пересічні лінії не мають логічного зв'язку між собою, тому зміни напрямку в точках перетину не допускаються. Дві або більше вхідні лінії можуть об'єднуватися в одну вихідну лінію. Якщо дві або більше лінії об'єднуються в одну лінію, місце об'єднання повинне бути зміщено (вони не повинні об’єднуватися в одній точці). При необхідності лінії в схемах слід розривати для уникнення зайвих перетинів або занадто довгих ліній, а також, якщо схема складається з декількох сторінок.

Лінії в схемах повинні підходити до символу або зліва, або зверху, а виходити або праворуч, або знизу. Лінії повинні бути спрямовані до центру символу.

### 1.3. Види блок-схем алгоритмів

За видом блок-схеми алгоритму розрізняють:

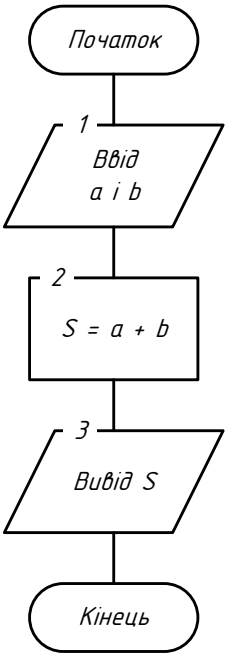



*Рис. 0.1 Блок-схема лінійного алгоритму*

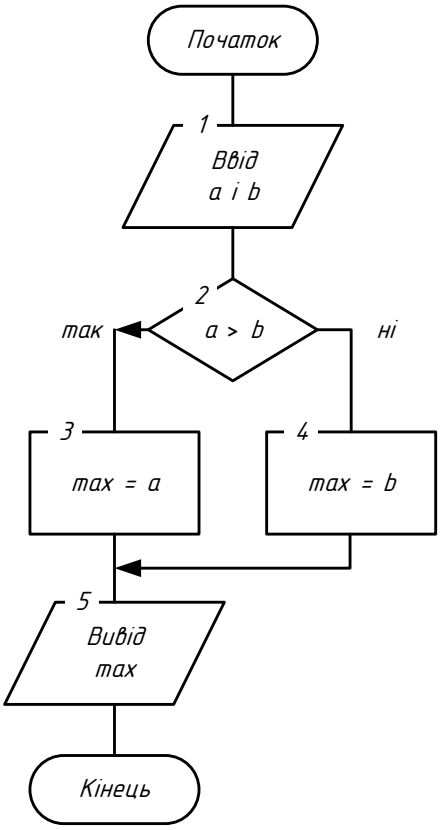



*Рис. 0.2 Блок-схема алгоритму з розгалуженням*

- *Алгоритм лінійної структури* – це такий алгоритм, де всі дії виконуються послідовно одна за однією і лише один раз. Блок-схема такого алгоритму наведена на рис. 0.1. Зображено алгоритм обчислення суми двох чисел $a$ і $b$ , які вводяться з клавіатури і виведення результату $S$ на екран.
- *Алгоритм з розгалуженням* – алгоритм, в якому передбачено розгалуження послідовності дій залежно від перевірки умови. Для прикладу наведено блок-схему алгоритму для знаходження максимального серед двох чисел $a$ і $b$ (рис. 0.2). Залежно від виконання умови $a>b$ змінній $max$ присвоюється значення $a$. Якщо умова не виконується, змінній $max$ присвоюється значення $b$.
- *Алгоритм циклічної структури* – алгоритм, що передбачає багаторазове повторення одних і тих ж дій над вихідними даними. До циклічних алгоритмів зводиться більшість методів обчислень, перебору варіантів. Для прикладу наведемо блок-схему алгоритму обчислення факторіалу числа $n$ – $n!$ (рис. 0.3). Також можна зустріти альтернативний варіант виконання блок-схеми (рис. 0.4).

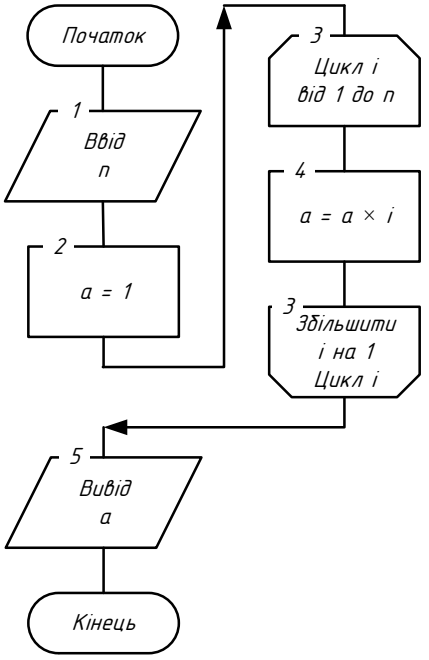



*Рис. 0.3 Блок-схема алгоритму циклічної структури*

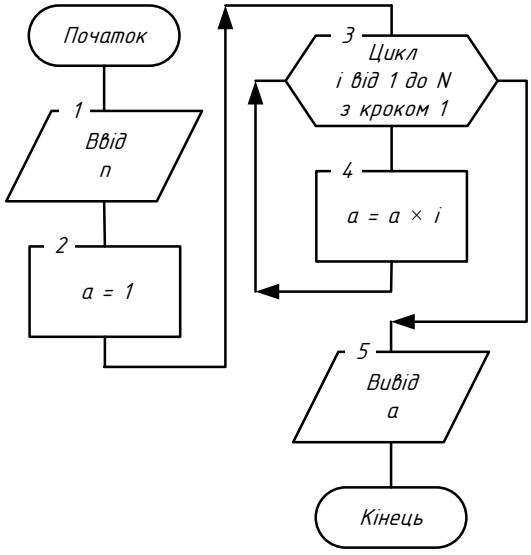



*Рис. 0.4 Блок-схема алгоритму циклічної структури (альтернативний варіант)*

Поширеною і хибною практикою є спроба використання блок-схем для ілюстрації алгоритму на низькому рівні (на рівні коду) – тобто, спроба вписувати в символи фрагменти коду на будь-якій штучній мові. Такий підхід застосовується лише до програм, що організовані відповідно до структурного підходу, і не можуть відобразити, наприклад, алгоритм, який реалізується у взаємодії абстракцій при інших підходах.

## 1.4. Деякі поширені алгоритми

### 1.4.1. Обмін двох змінних місцями

Цей алгоритм використовується майже всюди. Найпростіший спосіб обміняти дві змінні *a* та *b* місцями – використати третю змінну *c*. Під змінною розуміємо деяке місце, що може зберігати будь-які значення. Абстрактно це можна уявити як контейнер, що може бути заповнений рідиною. Коли необхідно обміняти вміст двох контейнерів, знадобиться третій. Переливаємо:

1. з *a* тимчасово в *c*;
2. з *b* в *a*;
3. з *c* назад в *b*.

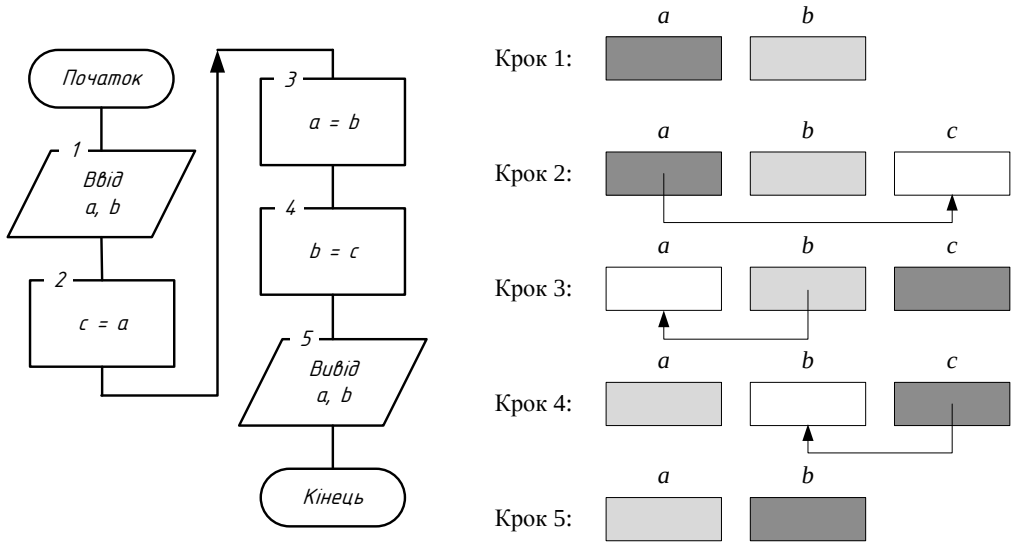



*Рис. 0.5 Алгоритм обміну двох змінних місцями*

### 1.4.2. Алгоритми сортування

*Сортуванням* називають впорядковування по ключах (тобто за якою-небудь ознакою) елементів деякої структури даних на якій визначено відношення порядку. Нехай існує послідовність $a_0$, $a_1$... $a_{n-1}$ і функція порівняння, яка на будь-яких двох елементах послідовності приймає одне з

трьох значень: менше, більше або рівне. Задача сортування полягає у перестановці елементів послідовності так, щоб виконувалася умова: $a_i <= a_{i+1}$, для всіх $a_i$ і від 0 до $n$-1.

Мабуть, жодна інша проблема не породила такої кількості найрізноманітніших рішень, як задача сортування. Чи існує певний універсальний, найкращий алгоритм? Маючи приблизні характеристики вхідних даних, можна підібрати метод, який працює оптимальним чином. Для того, щоб обґрунтовано зробити такий вибір, розглянемо параметри, по яких проводититься оцінка алгоритмів:

- *час сортування* – основний параметр, що характеризує швидкодію алгоритму;
- *пам'ять* – ряд алгоритмів вимагає виділення додаткової пам'яті під тимчасове зберігання даних. При оцінці пам'яті, що використовується, не враховується місце, яке займає початковий масив і незалежні від вхідної послідовності витрати, наприклад, на зберігання коду програми.

Сортування вибором

Ідея полягає у тому, щоб створювати відсортовану послідовність шляхом приєднання до неї у правильному порядку одного елемента за іншим.

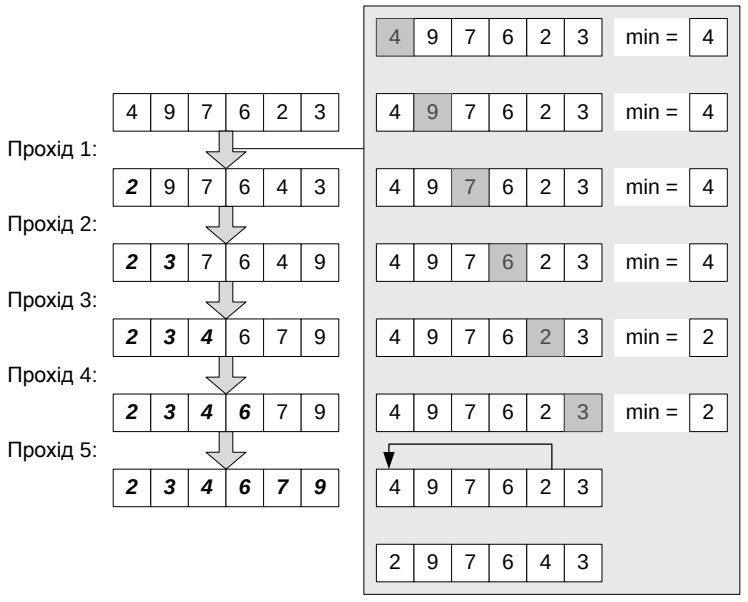



*Рис. 0.6 Сортування вибором*

Будуватимемо відсортовану послідовність, починаючи з лівого кінця масиву. Алгоритм складається з $n$-1 послідовних кроків, починаючи від

нульового і закінчуючи ($n$–2)-им. На $i$-му кроці вибираємо найменший (чи найбільший) з елементів $a_i$... $a_{n-1}$ і міняємо його місцями з $a_i$. Послідовність кроків при $n$=6 зображена на Рис. 0.6. Незалежно від номера поточного кроку $i$, послідовність $a_0$... $a_i$ є відсортованою. Таким чином, на ($n$–2)-у кроці вся послідовність, окрім $a_{n-1}$ виявляється відсортованою, а $a_{n-1}$ стоїть справедливо на останньому місці – всі менші елементи вже переміщені вліво.

Основні кроки алгоритму:

1. знаходимо позицію мінімального (чи максимального) елементу в поточному масиві;
2. здійснюємо обмін цього елементу із значенням першого невідсортованого елементу;
3. сортуємо решту масиву, виключивши з розгляду вже відсортовані елементи;

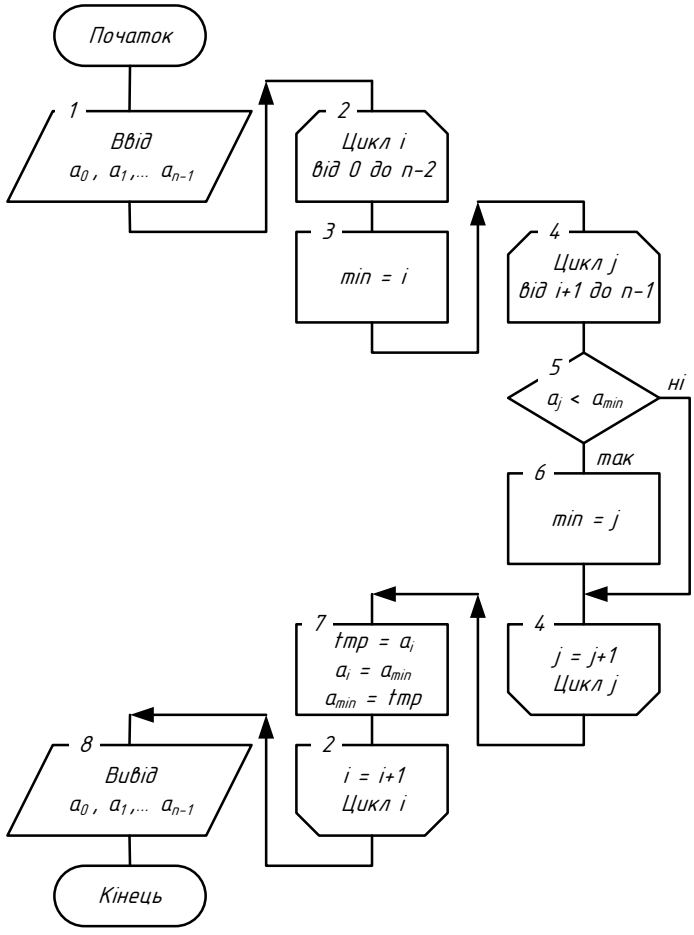



*Рис. 0.7 Блок-схема алгоритму сортування вибором*

Для знаходження найменшого елементу з $n$, які розглядаються, алгоритм виконує $n$-1 порівнянь. Із врахуванням того, що кількість елементів, які розглядаються, на черговому кроці зменшується на одиницю, загальна кількість операцій: $(n\text{-}1) + (n\text{-}2) + (n\text{-}3) + ... + 1 = 1/2 \times ( n^2+n ) = O(n^2)$. Таким чином, оскільки кількість обмінів завжди буде менша кількості порівнянь, час сортування росте квадратично відносно кількості елементів. Алгоритм не використовує додаткової пам'яті. Блок-схема наведена на рис. 0.7.

Бульбашкове сортування

Принцип дій простий: обходимо масив від $a_0$ до $a_{n-1}$, попутно міняючи місцями невідсортовані сусідні елементи $a_i$, $a_{i+1}$. В результаті першого проходу на останнє місце "спливе як бульбашка" максимальний (чи мінімальний) елемент. Тепер знову обходимо невідсортовану частина масиву (від $a_0$ до $a_{n-2}$) і міняємо по шляху невідсортованих сусідів. Другий необхідний елемент виявиться на передостанньому місці. Продовжуючи в тому ж дусі ($n$–2) раз, обходимо все меншу невідсортовану частину масиву, неявно вставляючи знайдені максимуми (чи мінімуми) в кінець (рис. 0.8).

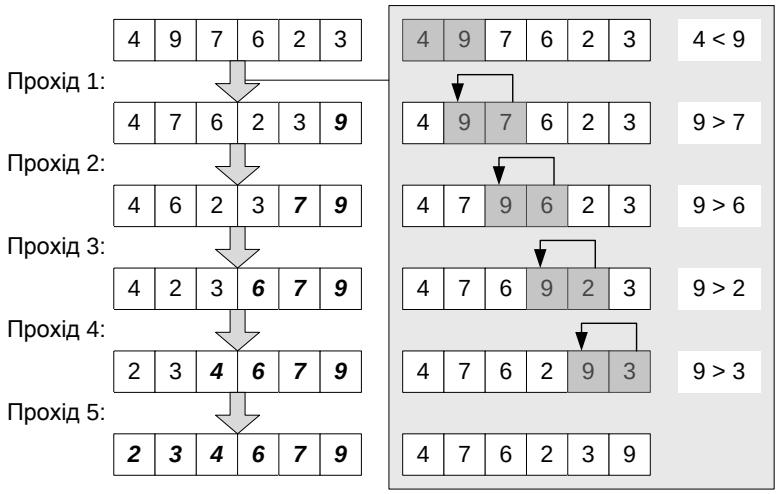



*Рис. 0.8 Бульбашкове сортування*

Кількість порівнянь і обмінів мають квадратичний порядок зростання: $O(n^2)$, додаткова пам'ять не потрібна. Блок-схема наведена на рис. 0.9.

Основні кроки алгоритму:

1. порівнюємо пари елементів у масиві;
2. здійснюємо обмін елементів місцями, якщо вони не відсортовані;
3. повторюємо попередні дії стільки разів, скільки є елементів, виключивши з розгляду вже відсортовані.

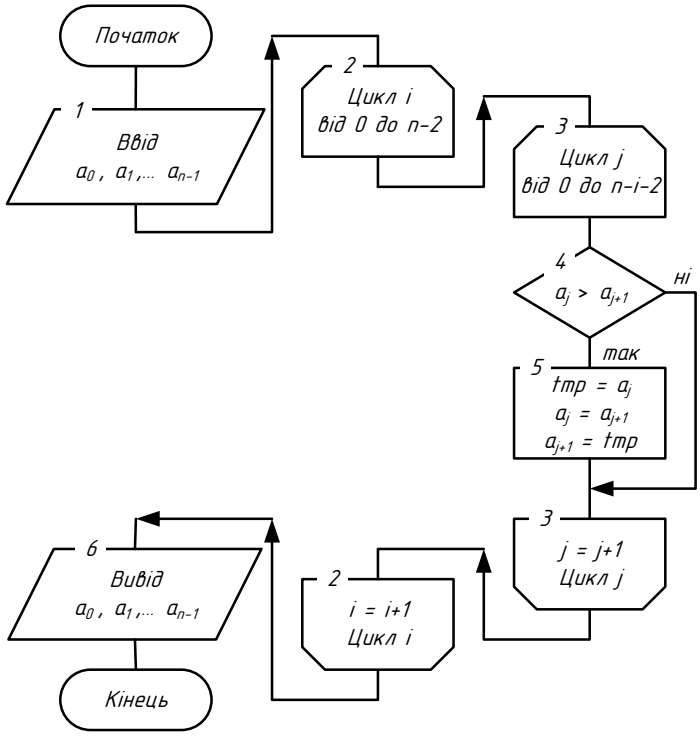



*Рис. 0.9 Блок-схема алгоритму бульбашкового сортування*

Сортування вставками

Алогоритм сортування простими вставками є дуже подібний на бульбашкове сортування. У ньому аналогічним чином робляться проходи по частині масиву, і аналогічним чином у його частині "зростає" відсортована послідовність. Проте у бульбашковому сортуванні або сортуванні вибором можна було чітко заявити, що на *i*-му кроці частина елементів вже відсортована і більше нікуди не переміститься. У даному алгоритмі подібне твердження буде більш слабшим: послідовність $a_0,\ldots a_i$ впорядкована. При цьому у процесі функціонування алгоритму в неї вставлятимуться нові невідсортовані елементи.

Розберемо алгоритм, розглядаючи його дії на *i*-му кроці. Послідовність до цього моменту розділена на дві частини: відсортовану $a_0,\ldots a_i$ та невідсортовану $a_{i+1},\ldots a_{n-1}$. На наступному, ($i$+1)-му кроці алгоритму беремо $a_{i+1}$ і вставляємо на потрібне місце у відсортовану частину масиву. Пошук відповідного місця для чергового елемента вхідної послідовності здійснюється шляхом послідовних порівнянь з елементом, що стоїть перед ним. Залежно від результату порівняння елемент або залишається на поточному місці (вставка завершена), або вони міняються місцями і процес повторюється (рис. 0.10).

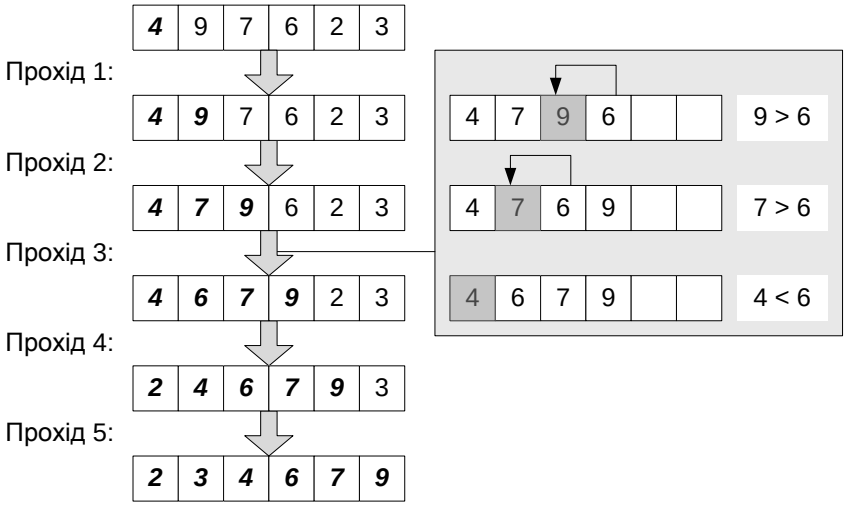



*Рис. 0.10 Алгоритм сортування вставками*

Аналогічно попереднім алгоритмам, складність оцінюється як $O(n^2)$, додаткова пам'ять не використовується. Блок-схема наведена на рис. 0.11.

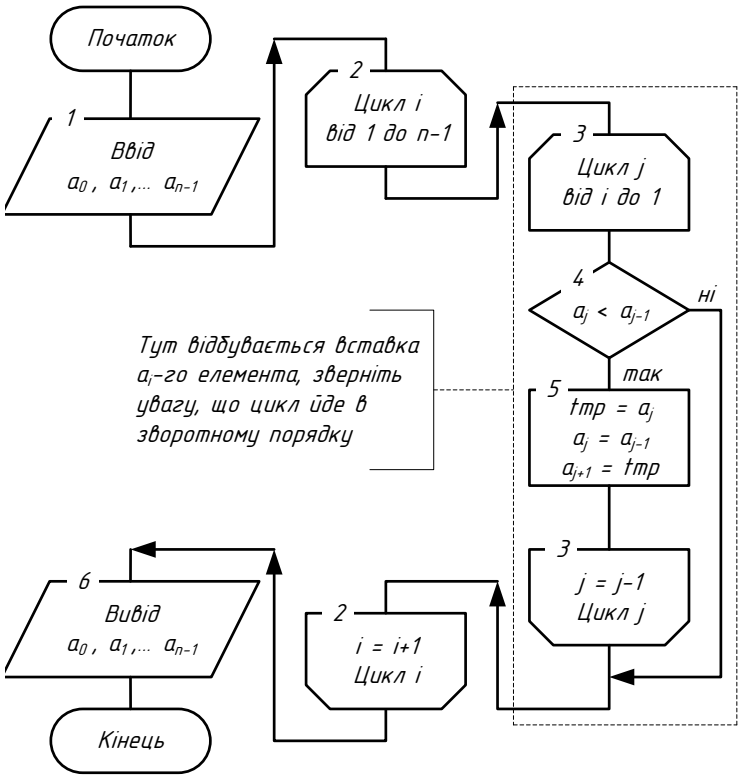



*Рис. 0.11 Блок-схема алгоритму сортування вставками*

Основні кроки алгоритму:

1. послідовно міняємо місцями новий елемент з попереднім відсортованим, починаючи з кінця, якщо він більший (чи менший);
2. якщо попередній відсортований не більший (чи не менший), завершуємо вставку;
3. продовжуємо, поки залишилися невідсортовані елементи.

### 1.4.3. Алгоритми пошуку

Однією з найважливіших дій зі структурованою інформацією є пошук [2]. *Пошук* – процес знаходження конкретної інформації в раніше створеній множині даних. Зазвичай дані представляють собою записи, кожен з яких має хоча б один ключ. *Ключ пошуку* – це поле запису, за значенням якого відбувається пошук. Пошук є однією з найбільш поширених дій в програмуванні. Існує багато різних алгоритмів пошуку, які принципово залежать від способу організації даних. У кожного алгоритму пошуку є свої переваги і недоліки. Тому важливо вибрати той алгоритм, який найкраще підходить для вирішення конкретного завдання.

Нехай існує послідовність $a_0$, $a_1$... $a_{n-1}$ і функція порівняння, яка на будь-яких двох елементах послідовності приймає одне з трьох значень: менше, більше або рівне. Задача пошуку полягає у знаходженні елементу послідовності $a_i$ так, щоб виконувалася умова: $a_i = ak$, де $ak$ – деякий наперед заданий ключ пошуку. Тобто, якщо $a_k$ входить в $a_0$, $a_1$... $a_{n-1}$, то знайдемо номер цього елемента масиву – визначимо перше входження заданого ключа.

Лінійний пошук

*Послідовний (лінійний) пошук* – це найпростіший вид пошуку заданого елемента на деякій множині, що здійснюється шляхом послідовного порівняння чергового розглянутого значення з шуканим до тих пір, поки ці значення не співпадуть. Ідея цього алгоритму полягає в наступному: множина елементів проглядається послідовно в деякому порядку, що гарантує, що будуть переглянуті всі елементи (наприклад, зліва направо). Якщо в ході перегляду множини буде знайдений шуканий елемент, перегляд припиняється з позитивним результатом; якщо ж буде переглянуто всю множину, а елемент не буде знайдений, алгоритм повинен видати деякий негативний результат. Складність алгоритму лінійна – $O(n)$. Блок-схема алгоритму наведена на рис. 0.12.

Бінарний пошук

*Двійковий (бінарний) пошук* (також відомий як *метод половинного ділення* та *метод дихотомії*) – алгоритм пошуку елемента в відсортованому масиві, що використовує дроблення масиву на половини.

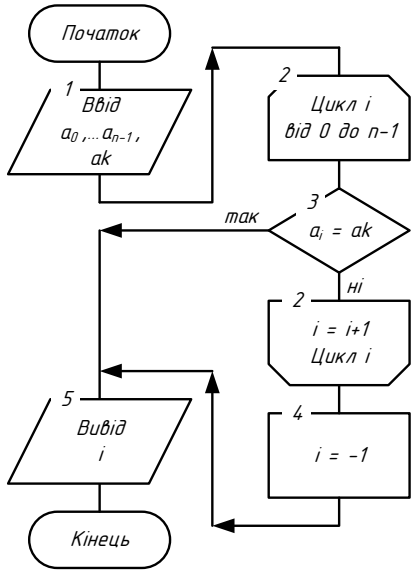



*Рис. 0.12 Блок-схема алгоритму лінійного пошуку*

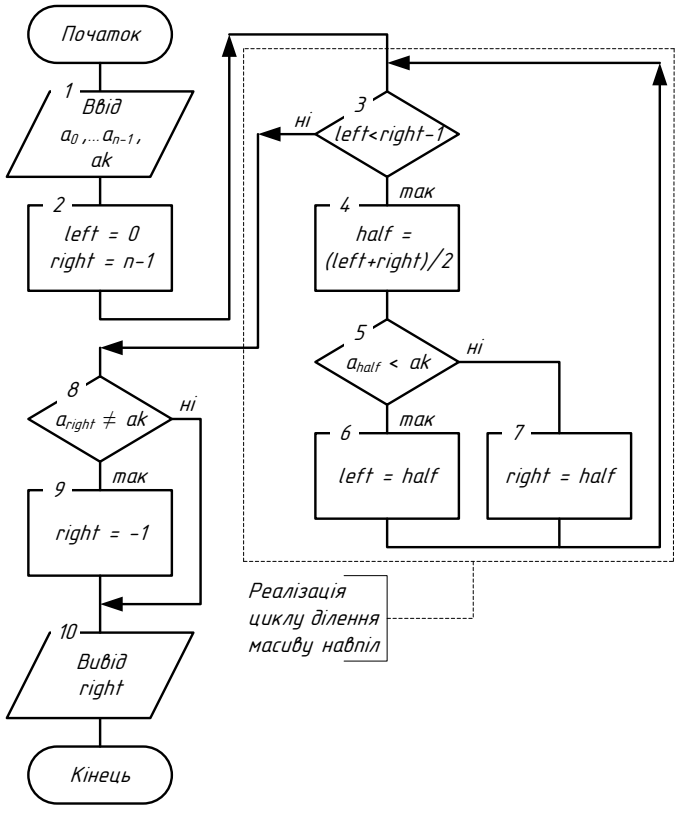



*Рис. 0.13 Блок-схема алгоритму бінарного пошуку*

Двійковий пошук полягає в тому, що на кожному кроці множина відсортованих елементів ділиться на дві частини і для розгляду залишається та частина множини, де знаходиться шуканий об'єкт. Складність алгоритму логарифмічнаа – $O(\log n)$. Блок-схема алгоритму наведена на рис. 0.13.

### 1.4.4.Половинне ділення для розв’язку трансцендентних рівнянь

Описаний алгоритм бінарного пошуку можна застосувати для пошуку коренів нелінійних чи навіть трансцендентних рівнянь [4]. Таким чином, *алгоритм половинного ділення*, який в цьому контексті також називають *методом бісекції*, є одним з найпростіших числових методів для розв’язку трансцендентних рівнянь.

Трансцендентними називаються нелінійні рівняння загалного виду $f(x) = 0$, що містять тригонометричні $\sin x$, $\cos x$, $\operatorname{tg} x$, $\operatorname{ctg} x$,… або інші нелінійні функції, наприклад, логарифмічну $\log x$ або експоненціальну $e^x$. При цьому, для роботи алгоритму необхідно виконати дві умови:

- функція $f(x)$ повинна бути неперервною на деякому вибраному відрізку $[a,b]$;
- $f(a)$ та $f(b)$ повинні мати різні знаки.

Наприклад, $f(x) = 2x^3 + 12\sin^3(3x) + 5x + 7$, графік якої зображено на рис. 0.14, має корінь на відрізку [–0,5:0,5].

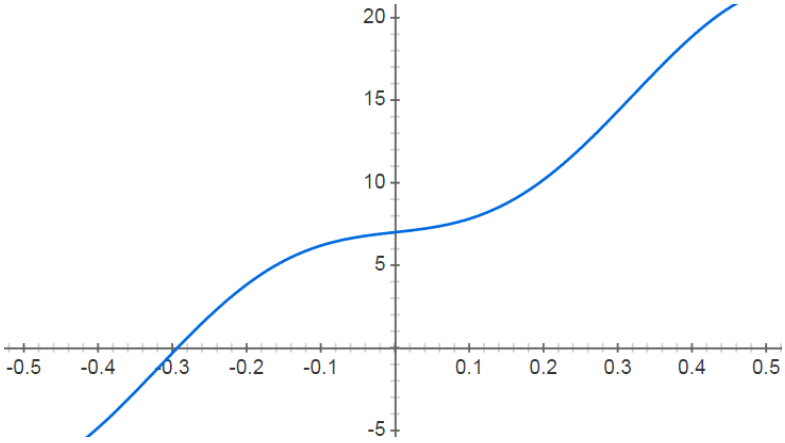



*Рис. 0.14 Графік монотонної трансцендентної функції*

Коли $f(a)$ і $f(b)$ мають протилежні знаки, знаходиться середина відрізку $c = (a + b) / 2$ та обчислюється значення функції в цій точці $f(c)$. Якщо знак $f(c)$ співпадає із знаком $f(a)$, то в подальшому замість $f(a)$ використовується $f(c)$, тобто на наступному кроці $a = c$ . Якщо ж $f(c)$ має знак, протилежний знаку

$f(a)$, тобто співпадає зі знаком $f(b)$, то наступному кроці $b = c$. Коли $f(b)$ достатньо близьке до 0, то процес обчислення закінчується. Як умову припинення ітераційного процесу часто найбільш доцільно використовувати умову $|b - a| \leq \varepsilon$, де $\varepsilon$ – задана похибка знаходження кореня. Блок-схема алгоритму представлена на Рис. 0.15.

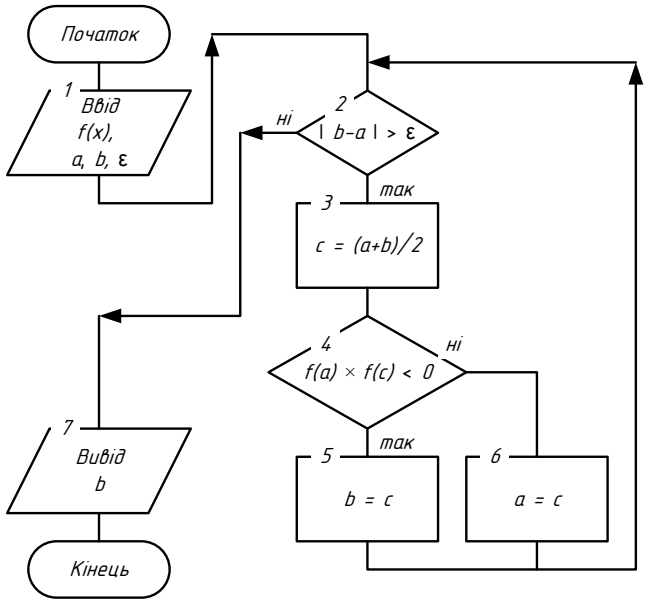



*Рис. 0.15 Блок-схема алгоритму половинного ділення для розв'язку трансцендентних рівнянь*

### 1.4.5. Числове інтегрування трансцендентних функцій

У найпростішому випадку *інтеграл* це деяка сума[1]. Якщо для визначеної та неперервної на відрізку $[a,b]$ функції $f(x)$ відома первісна $F(x)$, то означений інтеграл можна обчислити за відомою формулою Ньютона-Лейбніца:

$$\int_a^b f(x)dx = F(b) - F(a),$$

де $F'(x) = f(x)$. Проте, в багатьох випадках обчислити означений інтеграл за цією формулою неможливо, оскільки знайти первісну $F(x)$ через елементарні функції, як правило, не вдається. Навіть тоді, коли її можна визначити, вона часто має досить складний і незручний для обчислень вигляд. У таких випадках для обчислення означених інтегралів користуються числовими методами [4].

*Числове інтегрування* – це обчислення значення означеного інтеграла через ряд значень підінтегральної функції та її похідних. Найпростішими є формули,

[1] Саме позначення інтегралу ∫ ввів Лейбніц використавши першу літеру латинського слова *summa*.

які дають можливість наближено відшукувати значення інтеграла у вигляді лінійної комбінації кількох значень підінтегральної функції:

$$\int_a^b f(x)dx = \sum_{i=0}^{n-1} a_i f(x_i),$$

де $a_i$ – коефіцієнти формули (дійсні числа); $x_i$ – вузли формули. Коефіцієнти і вузли формули не повинні залежати від вибору функції $f(x)$. Наприклад, формули лівих, правих та середніх прямокутників:

$$I_{left} = \frac{b-a}{n-1}\left(f(x_0) + f(x_1) + \ldots + f(x_{n-1})\right) = \frac{b-a}{n-1}\sum_{i=0}^{n-1} f(x_i);$$

$$I_{right} = \frac{b-a}{n-1}\left(f(x_1) + f(x_2) + \ldots + f(x_n)\right) = \frac{b-a}{n-1}\sum_{i=1}^{n} f(x_i);$$

$$I_{middle} = \frac{b-a}{n-1}\left(f(x_1) + f(x_2) + \ldots + f(x_n)\right) = \frac{b-a}{n-1}\sum_{i=0}^{n-1} f\left(x_i + \frac{b-a}{2(n-1)}\right).$$

Величина $h = (b–a) / (n–1)$ – це *крок* числового інтегрування, чим він менший, тобто чим більша кількість вузлів $n$, тим точніший результат. Формули прямокутників "*збігаються*" до точного рішення з швидкістю $h$ (рис. 0.16).

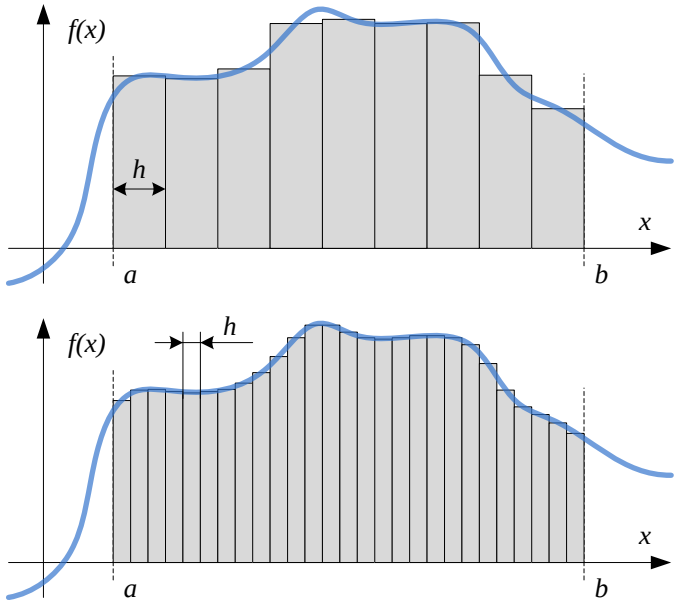



*Рис. 0.16 Числове інтегрування деякої функції за формулою середніх прямокутників*

Блок-схему алгоритму числового інтегрування трансцендентних функцій за формулою середніх прямокутників наведено на рис. 0.17.

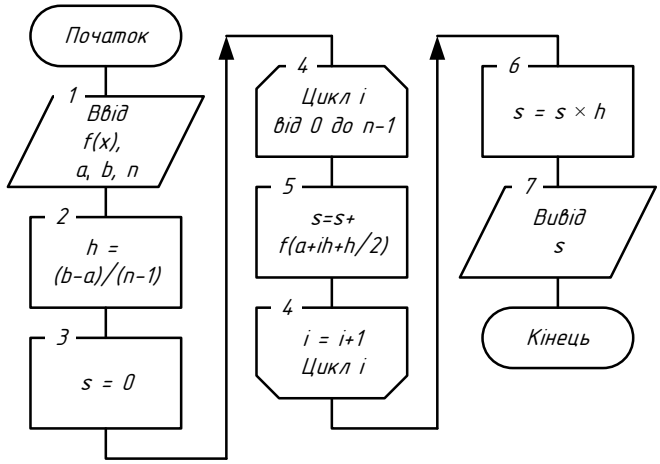



*Рис. 0.17 Блок-схема алгоритму числового інтегрування трансцендентних функцій за формулою середніх прямокутників*

## 1.5. Представлення інформації в комп'ютері

### 1.5.1.Системи числення

Проблема вибору системи числення для подання чисел у пам'яті комп'ютера має велике практичне значення. В разі її вибору звичайно враховуються такі вимоги, як надійність подання чисел при використанні фізичних елементів та економічність (використання таких систем числення, в яких кількість елементів для подання чисел із деякого діапазону була б мінімальною).

Сукупність прийомів та правил найменування й позначення чисел називається *системою числення*. Звичною і загальноприйнятою є позиційна десяткова система числення. Як умовні знаки для запису чисел вживаються цифри. Система числення, в якій значення кожної цифри в довільному місці послідовності цифр, що означає запис числа, не змінюється, називається *непозиційною*. Система числення, в якій значення кожної цифри залежить від місця в послідовності цифр у записі числа, називається *позиційною*.

Щоб визначити число, недостатньо знати тип і алфавіт системи числення. Для цього необхідно ще додати правила, які дають змогу за значеннями цифр встановити значення числа. Найпростішим способом запису натурального числа є зображення його за допомогою відповідної кількості паличок або рисочок. Таким способом можна користуватися для невеликих чисел.

Наступним кроком було винайдення спеціальних символів (цифр). У непозиційній системі кожен знак у запису незалежно від місця означає одне й те саме число. Добре відомим прикладом непозиційної системи числення є

римська система, в якій роль цифр відіграють букви алфавіту: I – один, V – п’ять, X – десять, C – сто, Z – п’ятдесят, D –п’ятсот, M – тисяча. Наприклад, 326 = CCCXXVI. У непозиційній системі числення незручно й складно виконувати арифметичні операції.

### 1.5.2.Позиційні системи числення

Загальноприйнятою в сучасному світі є десяткова позиційна система числення, яка з Індії через арабські країни прийшла в Європу. Основою цієї системи є число десять. *Основою системи числення* називається число, яке означає, у скільки разів одиниця наступного розрядку більша за одиницю попереднього.

Загальновживана форма запису числа є насправді не що інше, як скорочена форма запису розкладу за степенями основи системи числення, наприклад:

$$130678 = 1 \times 10^5 + 3 \times 10^4 + 0 \times 10^3 + 6 \times 10^2 + 7 \times 10^1 + 8 \times 10^0$$

Тут 10 є основою системи числення, а показник степеня – це номер позиції цифри в записі числа (нумерація ведеться зліва на право, починаючи з нуля). Арифметичні операції у цій системі виконують за правилами, запропонованими ще в середньовіччі. Наприклад, додаючи два багатозначних числа, застосовуємо правило додавання стовпчиком. При цьому все зводиться до додавання однозначних чисел, для яких необхідним є знання таблиці додавання.

Для зображення цілих чисел від 1 до 999 у десятковій системі достатньо трьох розрядів, тобто трьох елементів. Оскільки кожен елемент може перебувати в десятьох станах, то загальна кількість станів – 30, у двійковій системі числення: $999_{10}=1111100111_2$, необхідна кількість станів – 20 (індекс знизу зображення числа – основа системи числення).

У такому розумінні є ще більш економічна позиційна система числення – *трійкова*. Так, для запису цілих чисел від 1 до $10^9$ у десятковій системі числення потрібно 90 станів, у двійковій – 60, у трійковій – 57, що робить її оптимальнішою за двійкову. Але трійкова система числення не дістала поширення внаслідок труднощів фізичної реалізації.

Найпоширенішою для подання чисел у пам’яті комп’ютера є *двійкова система числення*. Для зображення чисел у цій системі необхідно дві цифри: 0 і 1, тобто достатньо двох стійких станів фізичних елементів. Ця система є близькою до оптимальної за економічністю, і крім того, таблички додавання й множення в цій системі елементарні:

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 10 |

| × | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

### 1.5.3. Біти та байти

Вся інформація, що зберігається та обробляється засобами обчислювальної техніки, незалежно від її типу (числа, текст, графіка, звук, відео тощо), представлена у двійковій системі. Знаки цієї системи 0 та 1 називають *бітами* (*bit, binary digit*). Одним бітом можна виразити два поняття: 0 або 1 (ні або так, хибне або істинне). Якщо кількість бітів збільшити до двох, то тоді можна закодувати чотири поняття : 00, 01, 10, 11. Трьома бітами кодують вісім понять. Збільшуючи на одиницю кількість розрядів в системі двійкового кодування, збільшуємо в два рази кількість значень, які можуть бути виражені в цій системі кодування, тобто кількість значень вираховується за формулою $N = 2m$, де $N$ – кількість незалежних значень, що кодуються, $m$ – розрядність двійкового кодування. Найменшою одиницею об'єму даних прийнято вважати *байт* – групу з 8 *бітів*[1]. Байтом можна закодувати, наприклад, один символ текстової інформації.

Наступними одиницями кодування є [5]:

| Десяткова система | | | | Двійкові (бі) префікси | | | |
|---|---|---|---|---|---|---|---|
| Назва | Скорочення | | Степінь | Назва | Скорочення | | Степінь |
| байт | Б | (B) | $10^0$ | байт | Б | (B) | $2^0$ |
| кілобайт | кБ | (kB) | $10^3$ | кібібайт | КіБ | (KiB) | $2^{10}$ |
| мегабайт | МБ | (MB) | $10^6$ | мебібайт | МіБ | (MiB) | $2^{20}$ |
| гігабайт | ГБ | (GB) | $10^9$ | гібібайт | ГіБ | (GiB) | $2^{30}$ |
| терабайт | ТБ | (TB) | $10^{12}$ | тебібайт | ТіБ | (TiB) | $2^{40}$ |
| петабайт | ПБ | (PB) | $10^{15}$ | пебібайт | ПіБ | (PiB) | $2^{50}$ |
| ексабайт | ЕБ | (EB) | $10^{18}$ | ексбібайт | ЕіБ | (EiB) | $2^{60}$ |
| зетабайт | ЗБ | (ZB) | $10^{21}$ | зебібайт | ЗіБ | (ZiB) | $2^{70}$ |
| йотабайт | ЙБ | (YB) | $10^{24}$ | йобібайт | ЙіБ | (YiB) | $2^{80}$ |

Не слід плутати гігабайти та гібібайти, останні є більшими. Річ у тому, що наведене розділення було стандартизоване не так давно і тому, навіть у вчених колах можна зустрітися з плутаниною в термінах. Цим активно користуються, наприклад виробники флеш накопичувачів та жорстких дисків, що чесно пишуть розмір в гігабайтах, а підключивши накопичувач до комп'ютера інформація про розмір подається заниженою, оскільки представлена в двійковому виді.

### 1.5.4. Алгоритми переведення чисел

Оскільки $2^3 = 8$, а $2^4 = 16$, то кожних три двійкових розряди зображення числа утворюють один вісімковий, а кожних чотири двійкових розряди – один шістнадцятковий. Тому для скорочення запису адрес та вмісту оперативної

[1] Насправді так було не завжди. Існують архаїчні архітектури, де байт містить 6, 32, або 36 біти.

пам’яті комп’ютера використовують шістнадцяткову й вісімкову системи числення:

| 10 | 2 | 8 | 16 | 10 | 2 | 8 | 16 |
|---|---|---|---|---|---|---|---|
| 0 | 0000 | 0 | 0 | 8 | 1000 | 10 | 8 |
| 1 | 0001 | 1 | 1 | 9 | 1001 | 11 | 9 |
| 2 | 0010 | 2 | 2 | 10 | 1010 | 12 | a |
| 3 | 0011 | 3 | 3 | 11 | 1011 | 13 | b |
| 4 | 0100 | 4 | 4 | 12 | 1100 | 14 | c |
| 5 | 0101 | 5 | 5 | 13 | 1101 | 15 | d |
| 6 | 0110 | 6 | 6 | 14 | 1110 | 16 | e |
| 7 | 0111 | 7 | 7 | 15 | 1111 | 17 | f |

В процесі відлагодження програм та в деяких інших ситуаціях у програмуванні актуальною є проблема переведення чисел з однієї позиційної системи числення в іншу. Якщо основа нової системи числення дорівнює деякому степеню старої системи числення, то алгоритм переводу дуже простий: потрібно згрупувати справа наліво розряди в кількості, що дорівнює показнику степеня і замінити цю групу розрядів відповідним символом нової системи числення. Цим алгоритмом зручно користуватися коли потрібно перевести число з двійкової системи числення у вісімкову або шістнадцяткову. Наприклад:

$$10110_2 = \overbrace{010}\overbrace{110} = 26_8\,,\ 1011100_2 = \overbrace{0101}\overbrace{1100} = 5C_{16}\,.$$

Переведення чисел з вісімкової або шістнадцяткової систем числення у двійковому відбувається за зворотнім правилом: один символ старої системи числення заміняється групою розрядів нової системи числення, в кількості рівній показнику степеня нової системи числення. Наприклад:

$$472_8 = \overbrace{100}\overbrace{111}\overbrace{010} = 100111010_2\,,\ B5_{16} = \overbrace{1011}\overbrace{0101} = 10110101_2\,.$$

Якщо основа однієї системи числення дорівнює деякому степеню іншої, то перевід тривіальний. У протилежному випадку користуються правилами переведення числа з однієї позиційної системи числення в іншу (найчастіше для переведення із двійкової, вісімкової та шістнадцяткової систем числення у десяткову, і навпаки).

Для переведення чисел із системи числення з основою $p$ в систему числення з основою $q$, використовуючи арифметику нової системи числення з основою $q$, потрібно записати коефіцієнти розкладу, основи степенів і показники степенів у системі з основою $q$ і виконати всі дії в цій самій системі. Очевидно, що це правило зручне при переведенні до десяткової системи

числення. Наприклад:

- з шістнадцяткової в десяткову:

$$92C8_{16} = 9 \times (10_{16})^3 + 2 \times (10_{16})^2 + C \times (10_{16})^1 + 8 \times (10_{16})^0 =$$
$$= 9 \times (16_{10})^3 + 2 \times (16_{10})^2 + 12 \times (16_{10})^1 + 8 \times (16_{10})^0 = 37576_{10}$$

- з вісімкової в десяткову:

$$735_8 = 7 \times (10_8)^2 + 3 \times (10_8)^1 + 5 \times (10_8)^0 =$$
$$= 7 \times (8_{10})^2 + 3 \times (8_{10})^1 + 5 \times (8_{10})^0 = 477_{10}$$

- з двійкової в десяткову:

$$110100101_2 = 1 \times (10_2)^8 + 1 \times (10_2)^7 + 0 \times (10_2)^6 + 1 \times (10_2)^5 +$$
$$+ 0 \times (10_2)^4 + 0 \times (10_2)^3 + 1 \times (10_2)^2 + 0 \times (10_2)^1 + 1 \times (10_2)^0 = 421_{10}$$

Для переведення чисел із системи числення з основою $p$ в систему числення з основою $q$ з використанням арифметики старої системи числення з основою $q$ потрібно:

- для переведення *цілої частини*: послідовно число, записане в системі основою $p$ ділити на основу нової системи числення, виділяючи остачі. Останні записані у зворотному порядку, будуть утворювати число в новій системі числення;
- для переведення *дробової частини*: послідовно дробову частину множити на основу нової системи числення, виділяючи цілі частини, які й будуть утворювати запис дробової частини числа в новій системі числення.

Цим самим правилом зручно користуватися в разі переведення з десяткової системи числення, тому що її арифметика для нас звичніша. Наприклад:

$$99{,}25_{10} = 1100011{,}01_2$$

- для цілої частини 99:

| 99 | 2 | | | | | |
|---|---|---|---|---|---|---|
| –98 | 49 | 2 | | | | |
| ***1*** | –48 | 24 | 2 | | | |
| | ***1*** | –24 | 12 | 2 | | |
| | | ***0*** | –12 | 6 | 2 | |
| | | | ***0*** | –6 | 3 | 2 |
| | | | | ***0*** | –2 | ***1*** |
| | | | | | ***1*** | |

- для дробової частини 0,25:

| | | |
|---|---|---|
| × | ***0*** | 25 |
| | **,** | 2 |
| | ***0*** | 5 |
| × | | 2 |
| | ***1*** | 0 |

## 1.6. Основи архітектури комп’ютера та організації його пам’яті

### 1.6.1. Архітектура фон Неймана

Для кращого розуміння, як реалізувати розроблені алгоритми на комп’ютері з допомогою програм, необхідно розуміти принципи його функціонування. У загальному випадку, комп'ютер складається з центрального процесора, пам'яті і пристроїв введення-виведення. Комп'ютер працює під управлінням певної програми, яка попередньо завантажується в пам'ять за допомогою завантажувача, що входить до складу операційної системи. Програма звертається до вмісту інших осередків пам'яті і пристроїв введення-виведення для отримання даних і збереження результатів роботи програми. Така модель архітектури комп’ютера називаєть архітектурою фон Неймана (рис. 0.18), що була розроблена *John von Neumann* (американський математик) та *Oskar Morgenstern* (американський економіст) у 1944-у році.

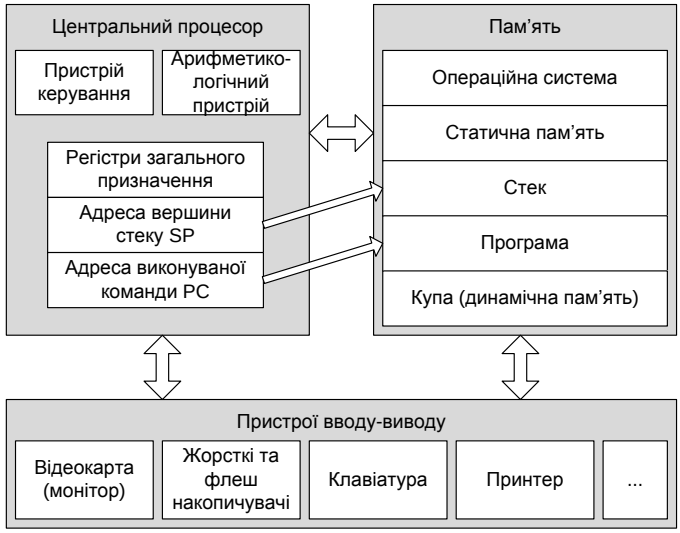



*Рис. 0.18 Модель архітектури комп’ютера фон Неймана (принстонська)*

Пам'ять комп'ютера розбита на комірки-байти (один байт зазвичай містить 8 біт), можна звертатися до комірки за її адресою – номеру байта в пам'яті. У комп'ютерах з 32-бітовим процесором максимальний обсяг пам'яті дорівнює $2^{32}$ = 4ГіБ, а з 64-бітовим – $2^{64}$. Дані в пам'яті зберігаються в кількох сусідніх байтах, наприклад, для зберігання цілих чисел потрібно 4 байти, а дійсних – 8 байт. Центральний процесор витягує чергову команду програми і всі необхідні дані для її виконання з пам'яті, виконує її і записує при необхідності результати назад в пам'ять. Дані для програми можуть розміщуватися в регістровій, статичній, стековій та динамічній пам'яті. Відмінність між цими видами пам'яті складається в способі адресації і тривалості зберігання даних.

### 1.6.2.Види пам’яті

| Пам'ять | Регістрова | Статична | Стекова | Динамічна |
|---|---|---|---|---|
| Обмеження | Від 4 до 32 значень | Обмежується розміром пам'яті комп'ютера | Кілька сотень кілобайт, можна змінити при компіляції | Обмежується розміром пам'яті комп'ютера |
| Адресація | Пряма: номер регістру визначається комп'ютером | Абсолютна: адреса комірки визначається на етапі компонування чи завантаження програми в пам'ять | Відносна: адреса комірки визначається додаванням значення стекового регістра до зміщення, що визначається на етапі компонування | Непряма: пам'ять виділяється під час виконання, адреса зберігається в комірках пам'яті інших видів |
| Тривалість зберігання даних | Визначається компілятором | Впродовж всього часу виконання програми | Впродовж виконання окремих функцій | Визначається програмою |
| Визначення у мові С | **`register`** | **`static`** | **`auto`** | **`malloc()`** **`free()`** |

Стек (*stack*) – область пам'яті, в яку додавання і видалення даних відбувається з одного боку. Стековий регістр SP вказує на вершину – першу зайняту комірку. При вході в блок програми дані додаються в стек, при виході – видаляються. Дані, що розміщуються в стеці, повинні мати фіксований розмір, так як зсув відносної адреси є константою і обчислюється під час компіляції. Також стек використовується для збереження значень регістрів, наприклад, при виклику підпрограми адреса поточної команди з регістра PC зберігається в стеку і в PC записується адреса викликаної підпрограми, а повернення з підпрограми полягає в завантаженні вершини стека назад в регістр PC.

Купа (*heap*) – дозволяє під час виконання програми створювати і знищувати дані довільного розміру в довільному порядку. Для повторного використання звільнення пам'яті підтримується список вільних областей пам'яті, в якому при виділенні пам'яті відбувається пошук підходящої за розмірами області, а якщо такий області не буде знайдено, запитується нова область пам'яті у операційної системи. Кількість дій для цього досить велика, тому виділення і звільнення пам'яті з купи є повільними операціями.

# ЛАБОРАТОРНА РОБОТА № 1.
# Процес розроблення програм на мові С в інтегрованому середовищі розробки програмного забезпечення Code::Blocks

## 1.1. Мета роботи

Ознайомитися з основними етапами створення програм. Ознайомитися з основними особливостями мови програмування на мові С. Ознайомитися з основними особливостями інтегрованого середовища розробки програмного забезпечення Code::Blocks. Навчитися складати, компілювати, компонувати та виконувати консольні програми.

## 1.2. Теоретичні відомості

### 1.2.1.Розроблення програм на мові С

Щоб отримати загальне уявлення про програмування, розділимо процедуру написання програми на сім основних етапів (рис. 1.1) [6]. Зауважимо, що це ідеалізація. На практиці, особливо в разі великих проектів, з'явиться необхідність переміщатися назад і вперед, використовуючи те, що отримано на більш пізньому етапі, для уточнення результатів, отриманих на більш ранній стадії.

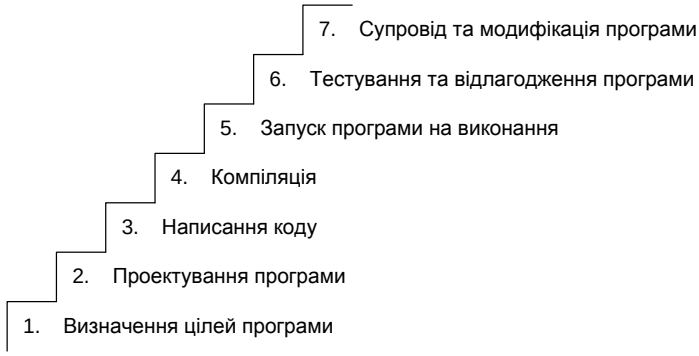



*Рис. 1.1 Сім основних етапів програмування*

Етап 1: Визначення цілей програми

Цілком природно, необхідно починати з чіткого бачення того, що програма повинна робити. Необхідно усвідомити якими є вхідні дані, які операції та маніпуляції необхідно над ними провести та якими будуть вихідні дані програми. На цьому етапі необхідно мислити в загальних термінах, а не в термінах якоїсь конкретної мови програмування.

Етап 2: Проектування програми

Після того, як стане зрозумілою концептуальна картина того, що програма повинна зробити, необхідно вирішити, як вона повинна це зробити. Яким

повинен бути користувальницький інтерфейс? Як повинна бути організована ця програма? Якими будуть потенційні користувачі? Скільки часу буде потрібно для завершення розроблення програми?

Необхідно вирішити, як представляти дані у програмі і, можливо, у допоміжних файлах, а також які методи слід використовувати для обробки даних. На початковому етапі вивчення програмування відповіді на ці питання не викличуть труднощів, але коли ситуація стане більш складною, то прийде розуміння, що ці рішення вимагатимуть врахування великої кількості обставин. Правильний вибір способу подання інформації може істотно полегшити розробку програми і обробку даних.

Підкреслимо ще раз, що на цьому етапі необхідно мислити загальними категоріями і не думати про конкретний програмний код, але деякі з рішень можуть бути засновані на загальних характеристиках мови програмування.

Етап 3: Написання коду

Тепер, коли проект програми створений, можна приступати до її реалізації, для чого необхідно написати програмний код. Саме на цій стадії будуть потрібні всі знання мови програмування. Динаміка процесу залежить від середовища програмування, яке використовується.

До числа робіт, що потрібно зробити на цьому етапі, відноситься документування виконаних дій. Найпростішим способом документування є коментування зі зрозумілими поясненнями, якими забезпечується програмний код.

Етап 4: Компіляція

Базова стратегія програмування полягає в тому, щоб використовувати програми, які перетворять вихідний код на мові програмування у виконуваний файл, який містить готовий до виконання програмний код на машинній мові. Ця робота виконується в два етапи: компіляція і компонування.

*Компілятор* – це програма, в обов'язки якої входить перетворення вихідного коду у виконуваний код, тобто компіляцію. Компілятор також перевіряє, чи не містить помилок програма. Коли компілятор знаходить помилки, він повідомляє про їх наявність і не створює виконуваний файл. Результатом роботи компілятора є проміжний код.

*Компонувальник* – програма, що комбінує цей проміжний код з іншими необхідними проміжними кодами, в результаті виходить виконуваний файл, що містить виконуваний код.

*Виконуваний код* – це команди на власній машинній мові конкретного комп’ютера, що безпосередньо ним виконуються.

Проміжний код зазвичай представляється у вигляді *об’єктного коду* – фактично виконуваного коду, який однак не може бути виконаний на конкретному комп’ютері, оскільки в ньому відсутні деякі важливі пункти.

Перший елемент, якого бракує в файлі об'єктного коду це код запуску, що представляє собою код, який діє в якості інтерфейсу між програмою і операційною системою. Наприклад, можна запускати програму на однакових комп'ютерах, один з яких функціонує під управлінням Windows, а інший – під управлінням Linux. В обох випадках обладнання одне і те ж, так що використовується один і той ж об'єктний код, але при цьому потрібні різні коди запуску для Windows і для Linux, оскільки ці системи підтримують програми по-різному. Другим відсутнім елементом є коди бібліотечних програм. Практично всі С-програми використовують стандартні бібліотечні функції. Об'єктний файл не містить код самих функцій, він просто містить команду, що вимагає використання цих функцій. Фактичний код зберігається в файлі, що отримав назву бібліотеки.

Роль компонувальника полягає в тому, щоб зібрати воєдино ці три елементи – об'єктний код, стандартний код запуску і бібліотечний код – з подальшим запам'ятовуванням в окремому файлі, який називається виконуваним. Що стосується бібліотечного коду, то компонувальник витягує тільки код, необхідний для функцій, що викликаються з бібліотеки (рис. 1.2).

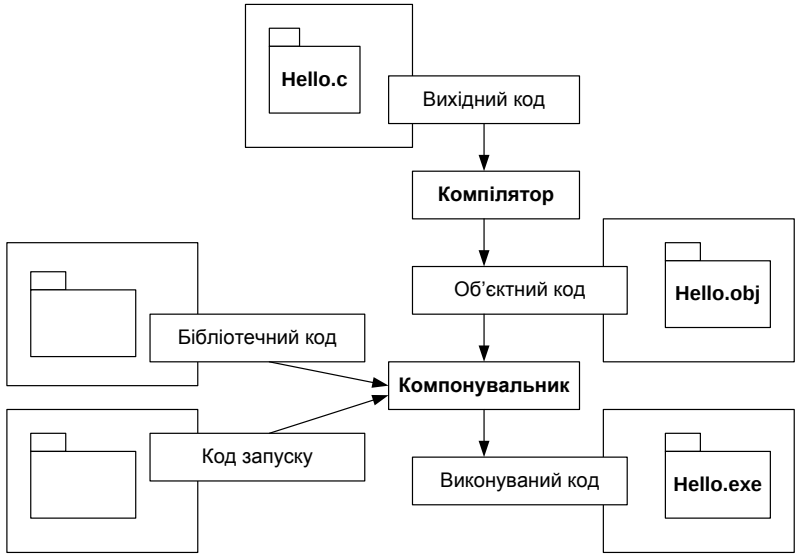



*Рис. 1.2 Компілятор і компонувальник*

В С використовується такий двоетапний підхід для модульної організації програм. З'являється можливість компілювати індивідуальні модулі окремо, а потім за допомогою компонувальника об'єднати скомпільовані модулі. Таким чином, якщо буде потрібно змінити якийсь один модуль, не потрібно буде повторно компілювати інші модулі. Крім того, компонувальник зв'язує програму з попередньо відкомпільованим бібліотечним кодом.

Етап 5: Запуск програми на виконання

Як правило, виконуваний файл є програмою, яку можна запускати на виконання. Щоб запустити програму, в багатьох відомих середовищах, включаючи консолі Windows, Unix, Linux, необхідно ввести з клавіатури ім'я виконуваного файлу. Інші середовища можуть зажадати введення команди запуску або використання будь-якого іншого механізму.

Середовища IDE (Integrated developtent environtents – *інтегроване середовище розробки,* ICP), подібні до тих, що поставляються для Windows і Macintosh, дають змогу редагувати і виконувати програми на С всередині середовища, вибираючи відповідні пункти меню або натискаючи спеціальні клавіші. Отримана програма може бути запущена на виконання безпосередньо з операційної системи шляхом одиночного або подвійного клацання на імені файлу або на відповідній піктограмі.

Етап 6: Тестування і налагодження програми

Той факт, що програма працює – хороший знак! Тим не менш, є ймовірність, що вона не працює належним чином. Звідси випливає, що необхідно переконатися, що програма робить саме те, що і повинна. Досить часто в програмах будуть виявлятися помилки.

*Відлагодження* – це процес виявлення і виправлення програмних помилок. Допущення помилок є природною складовою процесу навчання. Вони притаманні програмуванню. Існує багато можливостей зробити помилку. Можна зробити принципову помилку в проекті програми. Можна некоректно реалізувати хорошу ідею. Можна випустити з уваги неприпустимі вхідні дані. Можна неправильно використовувати конструкції самої мови програмування. Можна допускати помилки при наборі коду з клавіатури. Можна неправильно розставити дужки і так далі.

На щастя, ситуація небезнадійна, хоча можуть настати такі моменти, коли здасться, що це саме так. Компілятор відстежує багато видів помилок, крім того, можна зробити певні зусилля, щоб допомогти самому собі в пошуку помилок, що не відловив компілятор.

Інтегровані середовища розробки програмного забезпечення зазвичай містять засоби для покрокового виконання написаних програм з можливістю перегляду поточних значень змінних, тощо. Це дає змогу перевірити хід виконання програми і зрозуміти, що в ній відбувається не так, як задумано.

Етап 7: Супровід і модифікація програми

Коли програма створюється для себе або для кого-небудь ще, то, швидше за все, припускається, що вона буде використовуватися досить часто. Якщо це так, можливо з'являться причини для внесення в неї змін. Можливо існує якийсь незначний дефект, який проявляється при введенні імені, що починається з букв "Zz", або виникає бажання поліпшити щось в програмі. Або з часом необхідно

добавити в неї нову функціональну можливість. Можна адаптувати програму для виконання в різних комп'ютерних системах. Рішення задач подібного роду істотно спрощується, якщо програма була чітко документована а її код був написаний у відповідності до перевірених на практиці рекомендацій.

Програмування зазвичай не є таким послідовним процесом, яким є описаний вище процес. Час від часу прийдеться переміщатися туди і назад по етапах. Наприклад, коли пишеться програмний код, можна прийти до висновку, що обраний раніше план нездійсненний. Можна побачити кращий спосіб вирішення завдання. Можливо, після аналізу виконання програми виникне бажання змінити проектне рішення. Документування виконуваних дій допоможе переміщатися по етапах туди і назад.

Багато з тих, хто вивчає програмування нехтують етапами 1 і 2 (визначення цілей і побудова проекту програми) та переходять безпосередньо до етапу 3 (Написання програми). Перші написані програми будуть досить простими, щоб "прокрутити" весь процес розробки в голові. Якщо допускається помилка, її легко знайти. У міру того як програми стають все більшими і складнішими, уявне представлення програми починає підводити, а на виявлення помилок йде все більше часу. В кінцевому підсумку ті, хто знехтував стадією планування, приречені на марну витрату часу, на довгі години плутанини, до того ж отримуючи потворні, такі, що погано функціонують і важкі для розуміння програми. Чим більше і складніше завдання, тим більше часу доводиться витрачати на планування його рішення. Висновок, який випливає з усього вищесказаного, полягає в тому, що необхідно виробити в собі звичку складати плани, перш ніж приступати до написання власне коду.

### 1.2.2.Мова програмування С

*С* [7] – процедурна мова програмування загального призначення, розроблена у 1972 році у Bell Telephone Laboratories з метою написання нею операційної системи UNIX.

*Процедурне програмування* – це парадигма програмування, заснована на концепції виклику процедури. Процедури, також відомі як підпрограми, методи або функції. Процедури містять певну послідовність кроків до виконання. В ході виконання програми будь-яка процедура може бути викликана з будь-якого місця програми.

Протягом останніх десятиліть С стала однією з основних і найбільш широко поширених мов програмування. І хоча багато програмістів перейшли на більш претензійну об’єктно-орієнтовану мову С++, але мова С сама по собі все ще залишається важливою, особливо на шляху вивчення та переходу до С++.

У міру вивчення С можна переконатися, що вона має багато переваг, зокрема:

- компактний програмний код, відносно невеликі програми;
- кросплатформовість;
- швидкодія;
- наявність потужних структур управління;
- велика кількість прикладних програмних бібліотек найрізноманітнішого призначення.

Мова програмування С відрізняється мінімалізмом. Автори мови хотіли, щоб програми на ній легко переводилися в машинний код, щоб кожній елементарній складової програми після цього відповідало досить невелике число машинних команд. С створювалася з однією важливою метою: зробити простішим написання великих програм з мінімумом помилок за правилами процедурного програмування, не додаючи до коду програм нічого зайвого. Тому, С має наступні важливі особливості:

- просту мовну базу, з якої винесена в окремі бібліотеки велика кількість функціональних можливостей, наприклад математичні функції або функції управління файлами;
- орієнтацію на процедурне програмування;
- строгу систему типів змінних програм, що уберігає від безглуздих операцій;
- використання, так званого, препроцесора для, наприклад, визначення макросів і включення файлів з вихідним кодом;
- безпосередній доступ до пам'яті комп'ютера через використання вказівників на ділянки оперативної пам’яті;
- мінімальне число ключових слів;
- структури і об'єднання – визначені користувачем об’єднані типи даних, якими можна маніпулювати як одним цілим.

Мова С сильно вплинула на історичний розвиток мов програмування, що призвело до появи цілого класу, так званих, С-подібних мов.

Не зважаючи на те, що мова С розроблялася як мова "високого рівня" програмування, тобто така, що орієнтована в першу чергу на програмістів та дає їм гнучкі можливості доступу та маніпуляції ресурсами комп’ютера, за сучасними мірками вона є порівняно "низького рівня". У ній **немає:**

- прямих операцій над такими об’єктами як множини, стрічки, списки і масиви;
- операцій які маніпулюють цілими масивами або стрічками, натомість використовуються структури;
- засобів розподілу пам’яті окрім можливості визначення статичних змінних і стекового механізму при виділенні місця для локальних змінних функцій;

- засобів вводу-виводу і методів доступу до файлів.

Все це механізми високого рівня, які в мові С реалізуються за допомогою окремих бібліотечних функцій.

### 1.2.3. Інтегроване середовище розробки Code::Blocks

*Інтегроване середовище розробки* (ICP, англ. Integrated development environment, IDE) – комплексне програмне рішення для розробки програмного забезпечення. Зазвичай, складається з редактора початкового коду, інструментів для автоматизації складання та відлагодження програм. Більшість сучасних середовищ розробки мають можливість автодоповнення коду. Деякі середовища розробки містять у базовому комплекті компілятор та компонувальник. Деякі інтегровані середовища розробки містять систему керування версіями або інструменти для полегшення розробки графічного інтерфейсу користувача.

*Code::Blocks* – вільне кросплатформове середовище розробки програмного забезпечення [8], доступне за посиланням http://www.codeblocks.org/.

Існує також портативна версія з вбудованим компілятором MinGW (популярним варіантом GNU GCC Compiler для Windows), яка не потребує інсталювання на комп'ютері. Доступна за посиланням http://codeblocks.codecutter.org/.

Розглянемо алгоритм роботи з ICP[1]:

Перший запуск Code::Blocks

Середовище запускається з допомогою exe-файлу **CbLauncher.exe**. Якщо інсталювання пройшло без помилок, то після запуску Code::Blocks, під час його завантаження з'явиться заставка, на якій буде відображена версія. На рис. 1.3 наведено зображення заставки Code::Blocks де вказано номер збірки (версії).

При першому запуску з'явиться діалогове вікно зі списком знайдених компіляторів, з них треба вибрати той компілятор, який буде використовуватися за замовчуванням. Для виконання лабораторних робіт достатньо вибрати пропонований за замовчуванням GNU GCC Compiler і натиснути ***OK***[2].


*Рис. 1.3 Заставка Code::Blocks*

[1] На прикладі лінійки операційних систем Windows.

[2] Вказаний компілятор не є частиною ICP Code::Blocks і може інсталюватися на комп'ютері окремо. Доступний за посиланням http://www.mingw.org/.

На рис. 1.4 представлено діалогове вікно "Порада дня". Якщо необхідно, щоб воно не з'являлося при наступному запуску Code::Blocks, достатньо зняти галочку "Show tips at startup".

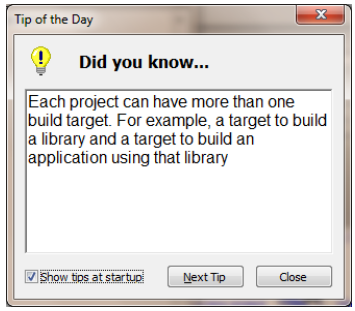



*Рис. 1.4 Вікно "Порада дня"*

Далі Code::Blocks запропонує використовувати себе в якості програми за замовчуванням для всіх файлів з розширенням **`.c`** / **`.cpp`** і **`.h`** / **`.hpp`**, які є на комп'ютері. Після вибору необхідної дії та натискання ***OK***, з'явиться головне вікно програми, представлене на рис. 1.5. Все, попереднє налаштування закінчено, можна приступати до роботи в Code::Blocks.

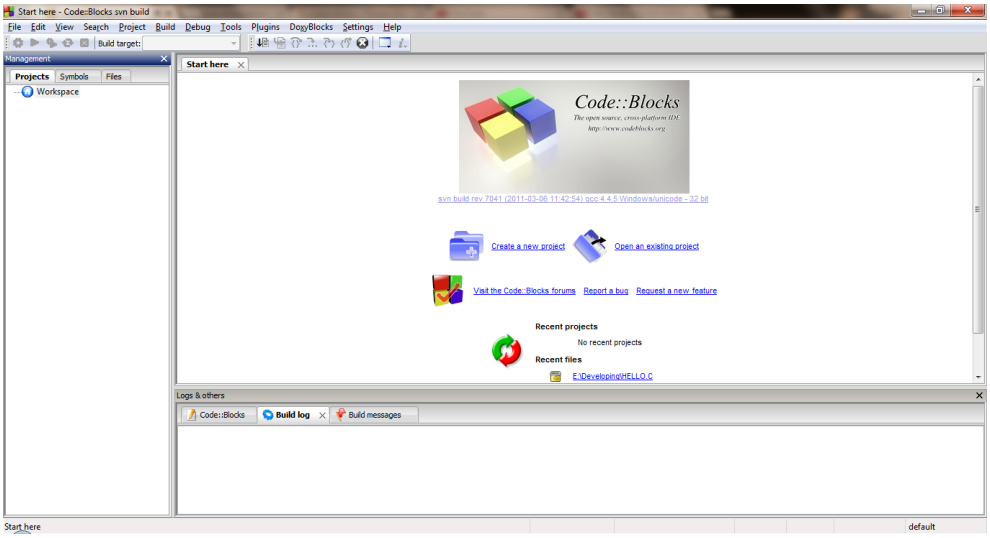



*Рис. 1.5 Основне вікно Code::Blocks*

Створення проекту в Code::Blocks на мові С

Під *проектом* розуміється деяка логічна організація файлів, необхідних для створення конкретного програмного продукту. Наприклад це можуть бути коди програм, піктограми з зображеннями іконок інтерфейсу користувача, файли налаштування середовища з спеціальними інструкціями для компілятора та

компонувальника, об'єктні файли, виконувані файли тощо. Середовище розробки програмного забезпечення організовує всі ці файли в окремі проекти.

На рис. 1.6 можна побачити вікно програми зі сторінкою "швидкого старту", що відкривається за замовчуванням при кожному запуску. ***Create a new project*** запускає майстер "створення проектів". ***Open an existing project***, відкриває діалогове вікно ***Open file***, в якому можна відкрити файл, вже існуючого проекту. Це ж діалогове вікно ***Open file*** можна викликати натиснувши поєднання клавіш ***Ctrl+O***. У списку ***Recent projects*** представлений список проектів, які були відкриті недавно. У списку ***Recent files***, представлений список файлів, що відкривалися недавно поза проектом.

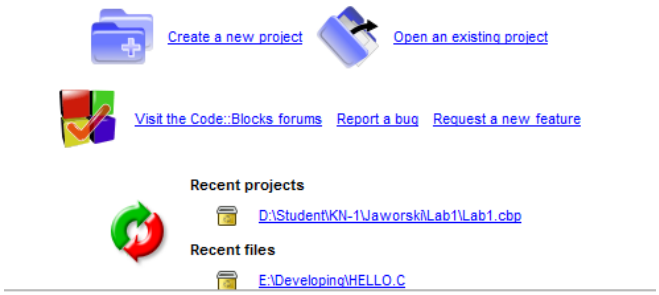



*Рис. 1.6 Швидкий старт*

Для створення проекту треба вибрати пункт меню ***File→New→Project ...*** або у вікні "швидкого старту" пункт ***Create a new project***, як показано на рис. 1.7.

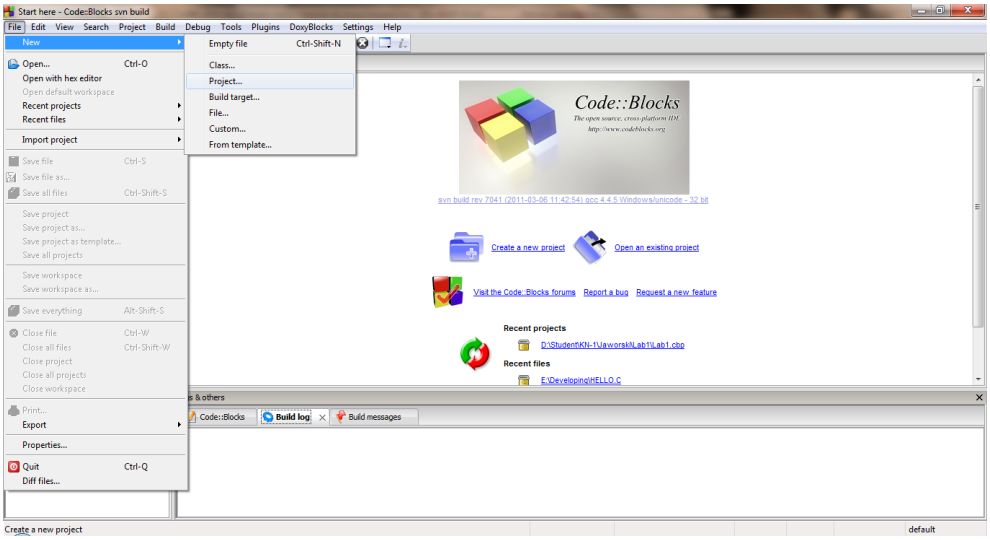



*Рис. 1.7 Загальний вид ІСР і відкритого меню **File→New***

Далі з'явиться діалогове вікно ***New from template***, в якому треба вказати шаблонпроекту.Для виконання лабораторних робіт треба вибрати ***Console application*** (рис. 1.8). І натиснути ***GO***, що викличе вікно майстра створення консольних програм.

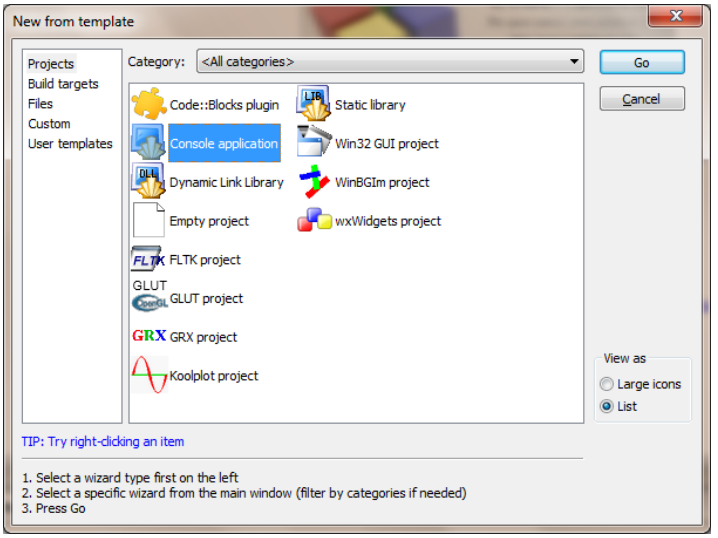



*Рис. 1.8 Діалогове вікно вибору шаблону проекту* ***New from template*** *додатків*

З'явиться діалогове вікно майстра з першою сторінкою рис. 1.9, де буде виведено привітання, і можливість поставити галочку ***Skip this page next time***, якщо потрібно, щоб наступного разу при створенні нового проекту ця сторінка майстра пропускалася. Для продовження слід натиснути ***Next***.

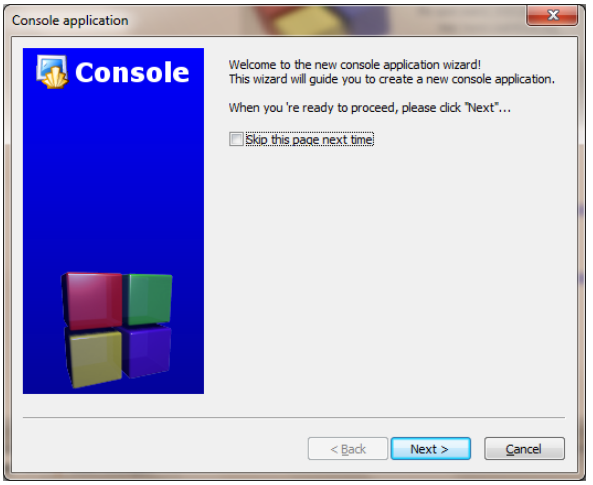



*Рис. 1.9 Діалогове вікно майстра з першою сторінкою "Ласкаво просимо в майстер нового консольного додатку"*

Відкриється сторінка (рис. 1.10), в якій потрібно вибрати якою мовою буде написана програма. Тут необхідно вибрати мову C та натиснути ***Next*** для переходу до наступної сторінки майстра рис. 1.11.

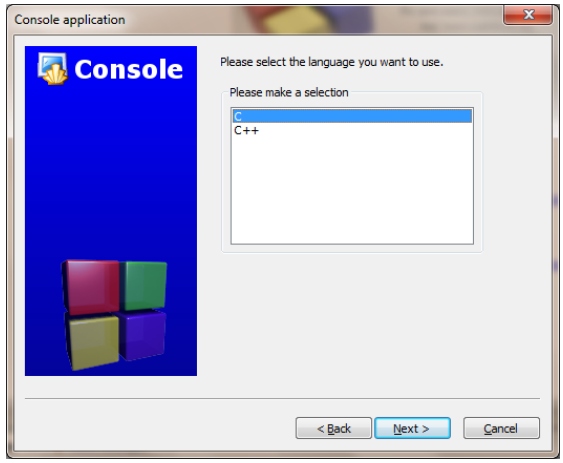



*Рис. 1.10 Сторінка майстра, де потрібно вибрати мову програмування*

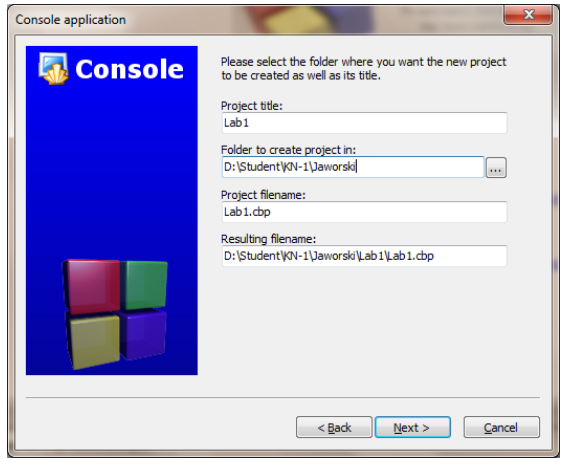



*Рис. 1.11 Сторінка майстра, де треба вказати ім'я створюваного проекту і каталог, де він буде розміщений*

Тут слід ввести в поле ***Project title*** ім'я проекту, в поле ***Folder to create project in*** вказати каталог, в якому буде зберігатися проект. Якщо проект створюється вперше, то треба створити каталог, де він і майбутні проекти будуть зберігатися. Для цього треба натиснути на кнопку з трикрапкою. Після цього відкриється вікно огляду катологів. Якщо вже існує каталог, де зберігаються лабораторні роботи, слід його вибрати, в іншому випадку треба його створити. Третє і четверте поля можна не заповнювати. Вказати ім'я файлу проекту можна в поле ***Project filename***. В поле ***Resulting filename*** вказується

кінцеве ім'я каталогу і проекту, отримане при заповненні попередніх полів. Після заповнення всіх необхідних полів слід натисути ***Next.***

Відкриється наступна сторінка (рис. 1.12). Тут треба вибрати компілятор, який буде використовуватися для компіляції програми (поле ***Compiler***) а також сценарії збірки. Сценарії допомагають отримати кілька версій однієї програми. Пропонуються два сценарії за замовчуванням, це ***Debug*** і ***Release***.

***Debug*** – сценарій компіляції, який використовується при відлагодженні програми. У цьому випадку компілятор додасть у виконуваний код програми спеціальні ділянки, що будуть використовуватися для покрокового виконання та можливості перегляду поточного значення змінних, тощо. Зазвичай це унеможливлює використання виконуваного модуля програми на інших комп’ютерах, тому слід розглядати цей варіант як чорновик.

***Release*** – сценарій компіляції для кінцевого результату програми, який можна передавати комусь у користування.

Якщо не потрібно створювати якийсь із цих сценаріїв, слід зняти відповідну галочку. Хоча б один з сценаріїв повинен бути обов'язково зазначений!

У кожному із сценаріїв пропонуються каталоги, куди будуть поміщатися файли, скомпільованої програми: ***Output dir*** – для всіх файлів, ***Objects output dir***– тільки для об'єктних файлів. У звичайних випадках ці каталоги міняти не потрібно. Після натискання на кнопку ***Finish*** проект буде автоматичний створений і відкритий[1].

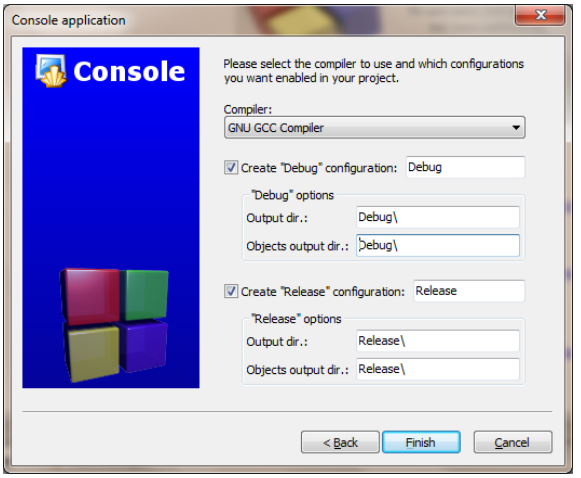



*Рис. 1.12 Сторінка майстра, де потрібно вибрати компілятор і визначити можливі сценарії збірки*

[1] В принципі, для маленьких програм створювати проект не обов’язково, однак це значно спрощує роботу. Можливості відлагодження доступні тільки при створенні проекту.

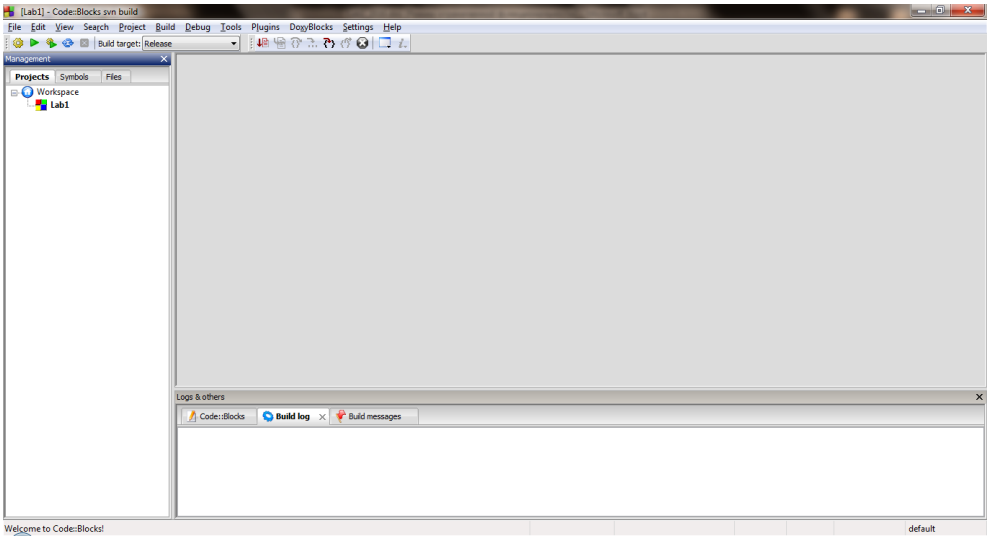



*Рис. 1.13 Головне вікно Code::Blocks*

Перші кроки в Code::Blocks після створення проекту

Після того, як було створено проект лабораторної роботи, з'явиться головне вікно з закритим редактором коду (рис. 1.13), де у вигляді дерева відображена ієрархічна структура проекту. Проект належить робочому простору ***Workspace***. Проект носить те ім'я, яке йому дали при створенні.

Необхідно створити файл з кодом програми, для цього слід виконати команду ***File→New→File ...***, після чого відкриється вікно створення файлів за шаблоном (рис. 1.14), де необхідно вибрати ***C/C++ source*** і натиснути ***GO***. З'явиться діалогове вікно майстра аналогічне до попереднього випадку.

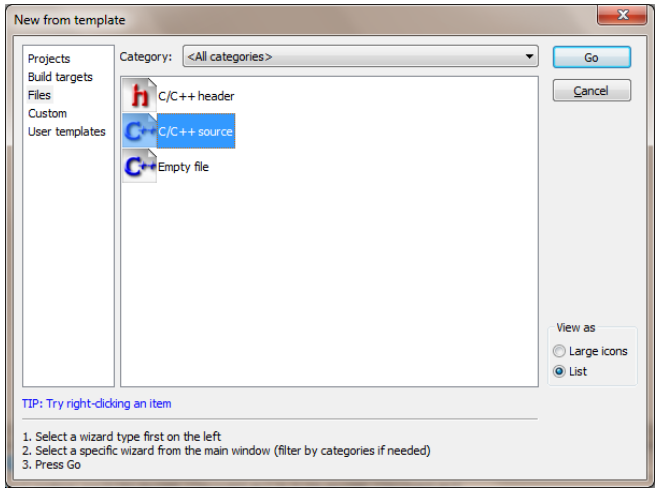



*Рис. 1.14 Діалогове вікно вибору шаблону файлу* ***New from template***

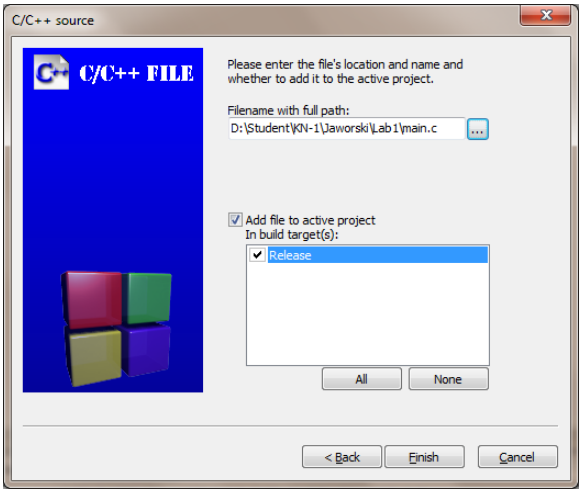



*Рис. 1.15 Сторінка майстра, де треба вказати ім'я створюваного файлу та приєднати його до проекту*

Виконуючи підказки майстра, необхідно вказати ім'я файлу та приєднати його до активного проекту (рис. 1.15). Після натискання кнопки ***Finish*** файл відкриється для редагування. Можна побачити, що дерево проекту у панелі ***Management*** також змінилося (рис. 1.16).

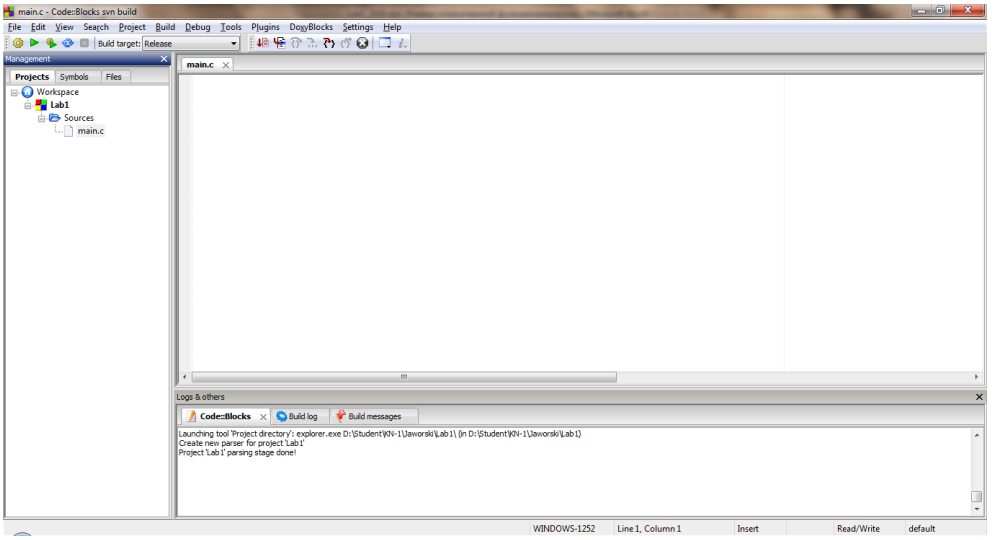



*Рис. 1.16 Головне вікно Code::Blocks з відкритим для редагування файлом*

На рис. 1.16, можна побачити панель, де відображаються заголовки всіх відкритих файлів. Якщо потрібно закрити якийсь файл, то слід натиснути на хрестик поруч з ім'ям цього файлу. Для закриття файлу, що відображається в даний момент, також можна скористатися поєднанням клавіш ***Ctrl+W.*** Для

переходу між відкритими файлами можна скористатися поєднанням клавіш ***Ctrl+Tab,*** або натискаючи на їх заголовки мишею. Якщо файл був змінений, то на його вкладці зліва від імені файлу з'являється зірочка. Для збереження файлу треба натиснути клавіші ***Ctrl+S,*** або мишкою в панелі інструментів натиснути на кнопку ***Save.***

У нижній частині зліва, в рядку стану, відображається шлях і ім'я відкритого в даний момент файлу.

За допомогою меню ***View*** (рис. 1.17) можна керувати зовнішнім виглядом Code::Blocks. Щоб відобразити або сховати панелі інструментів, треба зайти в меню ***View→Toolbars*** і відзначити відповідні панелі:

- ***Main*** – головна панель інструментів, на неї виводяться основні дії по роботі з проектами;
- ***Code completion*** – панель дозволяє переглядати об'єкти коду;
- ***Compiler*** – панель, на якій знаходяться кнопки управління компіляцією програми;
- ***Debugger*** – панель, на якій знаходяться кнопки управління відлагодженням програми.

Пункт меню ***View→Logs*** або клавіша ***F2*** відображають або ховають вікно повідомлень компілятора ***Logs*** внизу екрану.

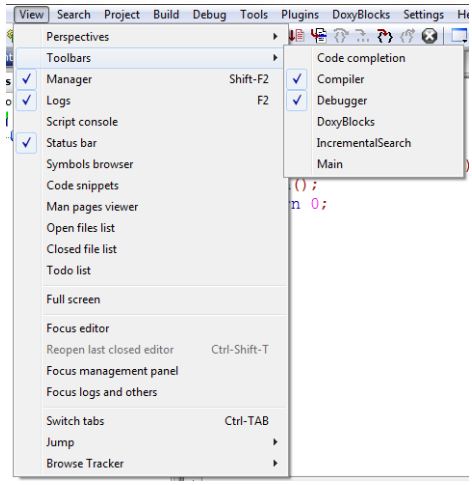



*Рис. 1.17 Меню **View**, де можна налаштувати зовнішній вигляд Code::Blocks*

Робота з більш, ніж одним відкритим проектом

Якщо відкрито одночасно декілька проектів (рис. 1.18), то той проект, який виділений жирним шрифтом, є активним проектом і незалежно від того, який файл відкритий в основному вікні в даний момент, компілюватиметься завжди буде саме активний проект. На малюнку він додатково виділений червоним кольором.

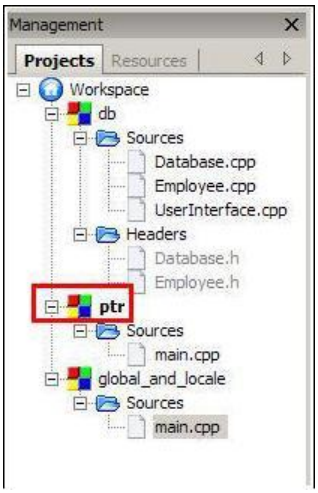



*Рис. 1.18 Панель **Management** с кількома відкритими проектами*

Для перемикання проектів слід навести курсор на той проект, який необхідно зробити активним і натисніть по ньому правою кнопкою миші. З'явитися меню, що випадає (рис. 1.19). У ньому слід вибрати пункт ***Activate project.*** Коли проект необхідно закрити, то слід вибрати пункт меню ***Close project.***

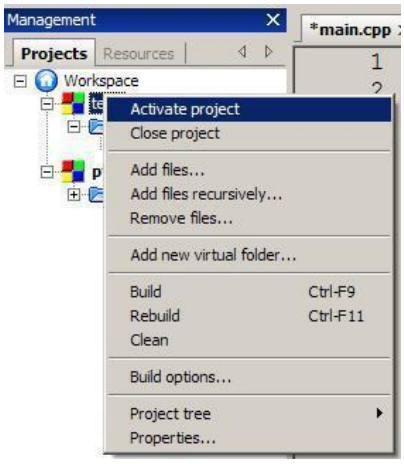



*Рис. 1.19 Меню проекту*

Компіляція та компонування проекту (лабораторної роботи)

Для компонування проекту, компіляції та запуску програми, треба натиснути клавішу ***F9*** або пункт в меню ***Build→Build and run*** (рис. 1.20). Якщо потрібно тільки скомпілювати проект без запуску, то слід вибрати пункт меню ***Build*** або натисніть клавіші ***Ctrl+F9.*** Якщо потрібно видалити тимчасові файли проекту, то треба вибрати пункт меню ***Clean.***

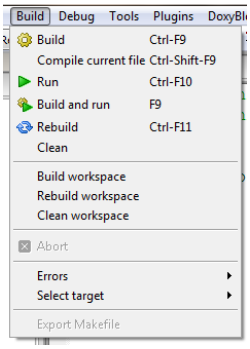



*Рис. 1.20 Меню* ***Build***

При створенні проекту було потрібно визначити сценарії збірки програми. Два сценарії збірки ***Debug*** або ***Release*** дають змогу отримати також два незалежних варіанти програми з різними опціями складання, і відповідно з різними опціями оптимізують додаток. Для того, щоб ними скористатися треба вибрати відповідний сценарій збірки. Це можна зробити в панелі інструментів ***Compiler***, де можна задати один з двох сценаріїв зі списку ***Build target.***

Змінити опції компіляції для кожного із сценаріїв можна у вікні ***Project build options*** (рис. 1.21), вибравши відповідний сценарій ***Debug*** або ***Release.*** Глобально ці налаштування задаються для всієї програми в вікні ***Compiler and debugger settings*** пункту меню ***Settings→Compiler and debugger ...***

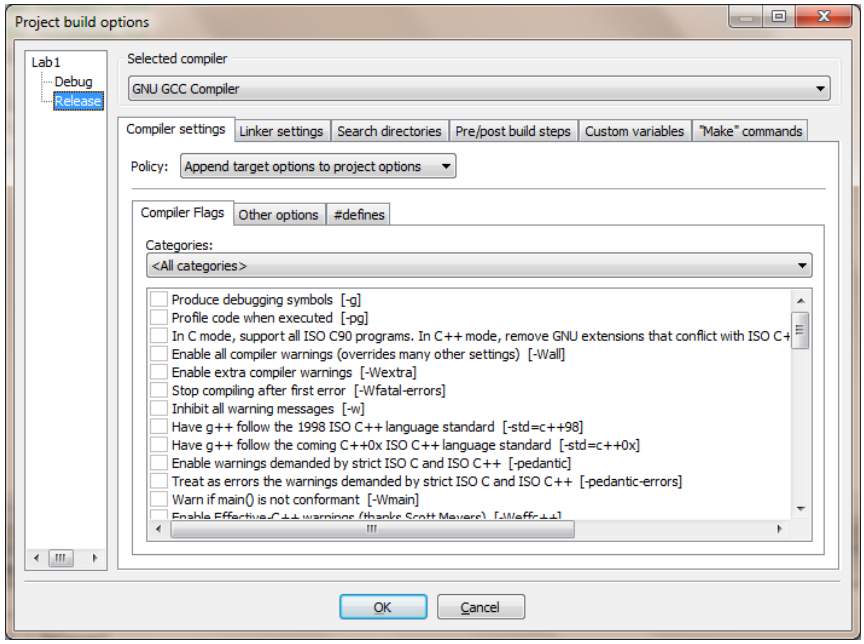



*Рис. 1.21 Вікно настройки компіляції і зв'язування проекту* ***Project build options***

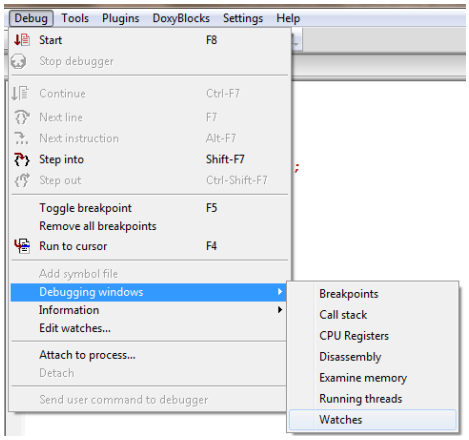



*Рис. 1.22 Меню Debug*

Відлагодження проекту (лабораторної роботи)

Для пошуку помилок виконання у програмі, тобто *багів* (історична назва з'явилася в 1940-их від англ. слова *bug*, або жук, комаха), використовується той відлагоджувач, що поставляється разом з компілятором. У даному випадку це Gdb. Для роботи з відлагоджувачем використовується меню ***Debug*** (рис. 1.22). Для запуску відлагоджувача потрібно вибрати пункт меню ***Debug→Start***. Після чого він запуститься, і якщо не вказано точок зупинки, то відлагоджувач відпрацює пройшовши по кроках всю програму та у випадку, коли не трапиться аварійна зупинка – завершиться. Для перегляду вмісту змінних, треба вказати точки зупинки, які відзначаються червоними крапками (рис. 1.23).

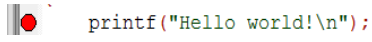



*Рис. 1.1 Приклад точки зупинки*

Точка зупинки встановлюється або натисканням лівої кнопки миші по сірій розділовій смузі поряд з номером рядка, або установкою курсору на рядок, в якому треба зупинитися. Відлагоджувач працює тільки у ***Debug*** збірці програми!

Для перегляду вмісту змінних і масивів використовується панель (вікно) спостереження змінних ***Watches*** (рис. 1.24).


*Рис. 1.24 Панель спостереження* ***Watches***

Щоб явно додати в це вікно змінні, за якими необхідно спостерігати, треба натиснути по вікну ***Watches*** правою кнопкою миші, після чого з'явиться меню, що випадає, де треба вибрати пункт ***Add watch***. Після цього відкриється діалогове вікно ***Edit watch*** (рис. 1.25), в якому в полі ***Keyword*** треба ввести ім'я змінної. Після чого натиснути кнопку ***OK.*** Якщо потрібно переглянути масив, то слід відзначити прапорцем ***Watch as array.***

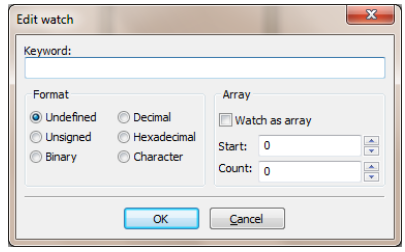



*Рис. 1.25 Діалогове вікно* ***Edit watch***

Вікно ***Watches*** автоматично відображатиме локальні змінні та аргументи функцій (рис. 1.26).

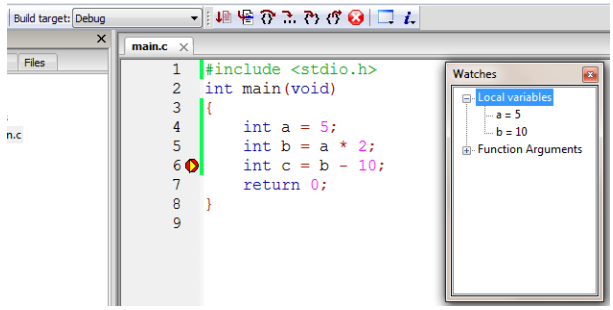



*Рис. 1.26 Приклад відлагодження програми*

### 1.2.4.Hello World – перша консольна програма

Консольна програма орієнтована на символьний ввід-вивід, що робить її корисною при вивченні стандартних функцій вводу-виводу мови С.

Програма *Hello world* – традиційна серед програмістів перша програма, що наводиться в підручниках з мов програмування. Програма виводить рядок "Hello, world!" або його еквіваленти.

Незважаючи на свою простоту, програма корисна тим, що дає змогу початківцю виконати всі дії, необхідні для написання, компілювання і запуску простої програми, написаної обраною мовою програмування.

На рис. 1.27 зображено код програми "Hello world!" на мові С[1].

[1] Увімкнути відображення номерів рядків можна у вікні ***General settings*** (***Settings→Editor…***) у вкладці ***Editor settings… / Other options / Show line numbers***.

```
1  #include <stdio.h>
2  #include <conio.h>
3  int main(void)
4  {
5      printf("Hello world!\n");
6      getch();
7      return 0;
8  }
9
```

*Рис. 1.27 Код програми "Hello world!" на мові С*

Перші два рядки це директиви препроцесора, що вказують необхідність підключення стандартних бібліотек. Бібліотека **<stdio.h>** тобто бібліотека стандартного вводу-виводу містить функцію **printf()**, що використовується для виводу текстових повідомлень на консоль. Бібліотека **<conio.h>**, тобто бібліотека консольного вводу-виводу містить функцію **getch()**, що відповідає за зчитування символу з клавіатури. Третій рядок оголошує головну функцію програми, яка завжди має назву **main**. З цієї функції починається будь-яка програма на мові С. Сьомий рядок містить оператор повернення **return**, який повертає число нуль після успішного завершення функції **main**, тобто після успішного завершення програми[1].

Для компілювання програми слід виконати команду ***Build***. Після цього компілятор створить у каталозі проекту об'єктний файл під назвою, що відповідає назві файлу з текстом програми, а компонувальник збере цей файл та інші необхідні об'єктні файли у виконуваний модуль під назвою, що відповідає назві проекту (рис. 1.28).

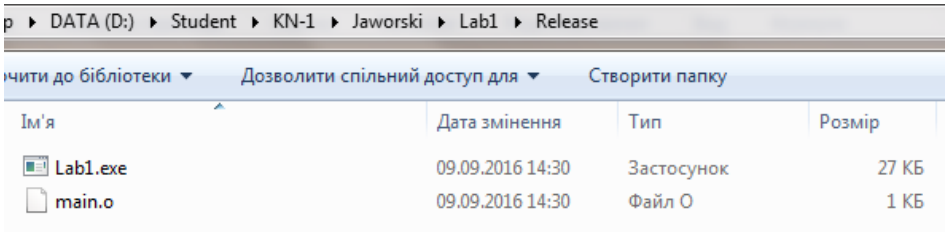



*Рис. 1.28 Результати компілювання та компонування програми*

На рис. 1.29 зображено результати виконання програми. В консольне вікно виводиться відповідне текстове повідомлення, після чого очікується введення будь-якого символу з клавіатури.

[1] Історично склалося так, що в разі успішного виконання програма та досить широке коло стандартних функцій повертають у місце свого виклику число нуль. У стандартній бібліотеці **<stdlib.h>** для цього спеціально відведена відповідна директива **EXIT_SUCCESS**. Така функціональна можливість використовувалася для аналізу ходу виконання програм до появи новіших інструментів, таких як наприклад обробники виняткових ситуацій.

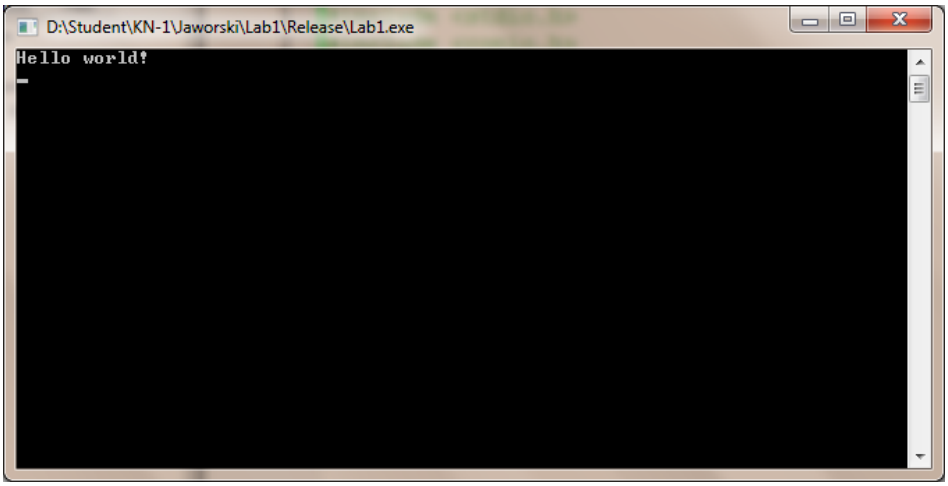



*Рис. 1.29 Результати виконання програми "Hello world!"*

### 1.2.5. Використання компілятора та компонувальника без IDE

На практиці трапляються випадки, коли необхідно скомпілювати та скомпонувати коди програми на комп'ютерах, де відсутні звичні інтегровані середовища розробки програмних продуктів. Наприклад в Linux системах компілятор мови С поставляється за замовчуванням, а для редагування тексту програм можна використати стандартний текстовий редактор.

Компілятор та компонувальник це також програми, що виконуються аналогічно до інших в конкретній операційній системі (інколи це одна і та ж програма, що виконує різні дії в залежності від вхідних параметрів). У Code::Blocks шляхи до виконуваних файлів компілятора, компонувальника та відлагоджувача можна редагувати глобально в вікні ***Compiler and debugger settings*** пункту меню ***Settings → Compiler and debugger ...*** у вкладці ***Toolchain executables*** (рис. 1.30)[1].

Виконуючи команду ***Build*** для проекту інтегроване середовище розробки програмного забезпечення засобами операційної системи по черзі запускає програми з вхідними параметрами, що відповідають назвам файлів проекту. Ці дії можна зробити вручну з консолі виконавши команди[2]:

```
set path=E:\CodeBlocks-EP\MinGW\bin\
cd D:\student\kn-1\jaworski\lab1
mingw32-gcc.exe main.c -o lab1.exe
lab1.exe
```

[1] У деяких випадках, для можливості використання функціональних можливостей нових стандартів мови С99 та новіших років необхідно буде явно вказати параметр компілятора "**`-std=c99`**" у вікні ***Compiler and debugger settings*** вкладці ***Compiler settings / Other options***.

[2] Наведено приклад консольних команд лінійки операційних систем Windows.

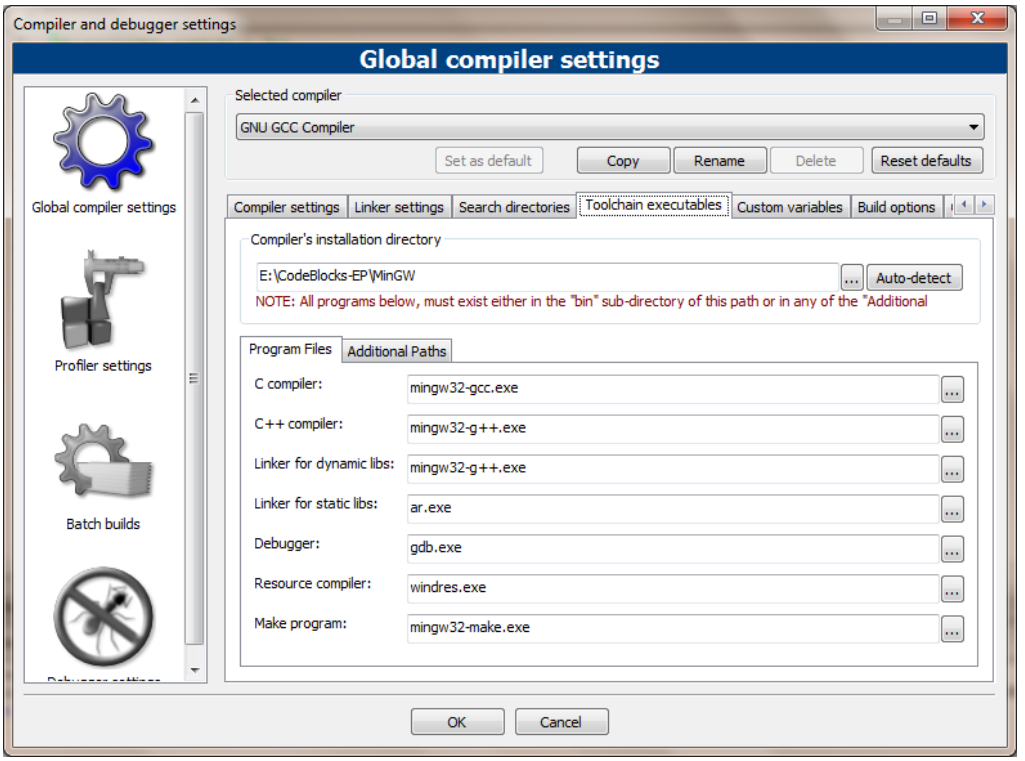



*Рис. 1.30 Вкладка* ***Toolchain executables***

У результаті отримаємо зібрану програму (рис. 1.31).

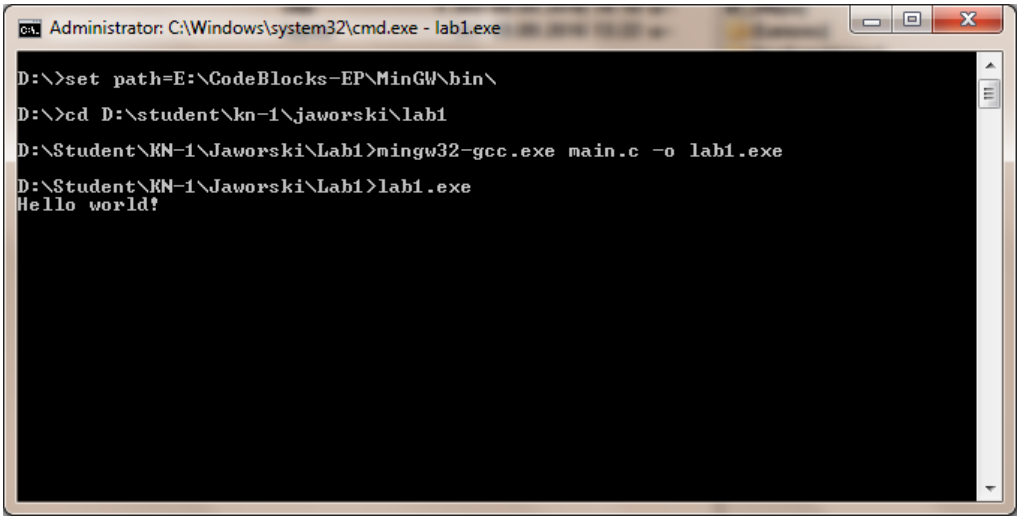



*Рис. 1.31 Компіляція та компонування програми "Hello world!"*
*з допомогою консольних команд операційної системи Windows та компілятора MinGW*

### 1.3. Контрольні запитання

1. Які є сім основних кроків для розроблення програм?
2. Що робить компілятор?
3. Що робить компонувальник?
4. Що робить відлагоджувач?
5. Чим об’єктний код відрізняється від виконуваного?
6. Що таке процедурне програмування?
7. Що таке інтегроване середовище розробки програмного забезпечення?
8. Які дії необхідно виконати для створення консольного проекту в інтегрованому середовищі розробки Code::Blocks?

### 1.4. Лабораторне завдання

1. Прослухати інформацію про порядок роботи в комп’ютерному класі. Проаналізувати інформацію на робочому столі Вашого ПК, запам’ятати порядок роботи (включення, ввід паролю, виключення).
2. Запустити середовище Code::Blocks, розглянути всі команди, запам’ятати найбільш необхідні для роботи команди.
3. У каталозі **`Student`** на диску **`D`** створити каталог з назвою своєї групи. В ньому створити каталог з назвою свого прізвища. В ньому створити консольний проект під назвою "**`Lab1`**". Усі назви не повинні містити в собі пробіли, крапки, коми та інші розділові знаки.
4. У проекті створити файл коду програми на мові С.
5. Написати код програми "Hello world!", скомпілювати, скомпонувати та запустити програму до виконання.
6. Модифікувати програму так, щоб вона виводила на консоль ваше прізвище.
7. Скомпілювати, скомпонувати та запустити на виконання програму з допомогою консольних команд.

### 1.5. Зміст звіту

1. Титульний аркуш.
2. Мета роботи.
3. Хід виконання лабораторного завдання.
4. Текст модифікованої програми.
5. Результати виконання модифікованої програми.
6. Аналіз результатів та висновки.

# ЛАБОРАТОРНА РОБОТА № 2.
# Основні поняття мови програмування С.
# Оператори розгалуження програми у мові С

## 2.1. Мета роботи

Вивчити основні поняття мови програмування С, операції, стандартні функції, оператори розгалуження програм.

## 2.2. Теоретичні відомості

### 2.2.1. Мова С[1]

Алфавіт мови С

Програма в мові С записується символами базового алфавіту (*basic character set*), який містить:

- 26 великих літер:
  **A B C D E F G H I J K L M**
  **N O P Q R S T U V W X Y Z**
- 26 маленьких літер:
  **a b c d e f g h i j k l m**
  **n o p q r s t u v w x y z**
- 10 десяткових цифр:
  **0 1 2 3 4 5 6 7 8 9**
- 29 спеціальних символів:
  **! " # % & ' ( ) * + , - . / :**
  **; < = > ? [ \ ] ^ _ { | } ~**
- символ пробілу, табулювання та переходу на новий рядок.

Із символів складаються базові елементи мови або лексеми (*tokens*), ними можуть бути:

- ключові слова (*key words*);
- ідентифікатори (*identifiers*);
- константи (*constants*);
- стрічкові літерали (*string literals*);
- знаки пунктуації (*punctuators*).

У свою чергу, лексеми є частиною виразів, а з виразів складаються інструкції та оператори. При компіляції програми на С з програмного коду виділяються лексеми максимальної довжини, що містять допустимі символи. Якщо в програмі є неприпустимий символ, то лексичний аналізатор (або компілятор) видасть помилку, і компіляція програми виявиться неможливою.

---

[1] Інформація відповідає стандарту C11 ISO/IEC 9899:2011 [7].

Ключові слова мови С

Мова містить наступні ключові слова, що є зарезервованими і не можуть бути використані у будь-якій іншій ролі (враховується регістр):

| | | |
|---|---|---|
| **auto** | **long** | **_Atomic** |
| **break** | **register** | **_Bool** |
| **case** | **restrict** | **_Complex** |
| **char** | **return** | **_Generic** |
| **const** | **short** | **_Imaginary** |
| **continue** | **signed** | **_Noreturn** |
| **default** | **sizeof** | **_Static_assert** |
| **do** | **static** | **_Thread_local** |
| **double** | **struct** | |
| **else** | **switch** | |
| **enum** | **typedef** | |
| **extern** | **union** | |
| **float** | **unsigned** | |
| **for** | **void** | |
| **goto** | **volatile** | |
| **if** | **while** | |
| **inline** | **_Alignas** | |
| **int** | **_Alignof** | |

Ідентифікатори мови С

Ідентифікатори використовуються для визначення змінних, функцій, структур, об'єднань, їх полів, перечислень, власних типів даних, міток, назв макросів та їх параметрів. Один і той ж ідентифікатор може визначати різні програмні об'єкти у різних ділянках програми в залежності від області його видимості (*scope of identifier visibility*).

Правила запису ідентифікаторів:

1. Ідентифікатори складаються з букв, цифр і символу підкреслення (“**_**”) (до складу ідентифікатора не може входити будь-який спеціальний символ).
2. Першим символом повинна бути буква або символ підкреслення. Однак, не рекомендується починати ідентифікатори із символу підкреслення оскільки багато змінних стандартних бібліотечних програм починається саме з цього символу.
3. Враховується регістр. Не можна плутати в ідентифікаторах великі і малі букви (**X** і **x** – це два різні ідентифікатори).
4. Немає обмежень на максимальну довжину ідентифікатора (однак, це залежить від конкретного компілятора і згідно стандарту максимальна довжина повинна становити хоча б 31 символ).

Константи мови С

Константи поділяються на:

- цілочисельні (*integer*)[1]:
  - десяткові:
    починаються з цифри, але не нуля;
    можуть містити символи:
    **0 1 2 3 4 5 6 7 8 9**
    наприклад: **1024**
  - вісімкові:
    починаються з префіксу нуль "**0**";
    можуть містити символи:
    **0 1 2 3 4 5 6 7**
    наприклад: **037**
  - шістнадцяткові:
    починаються з префіксу "**0x**" або "**0X**";
    можуть містити символи:
    **0 1 2 3 4 5 6 7 8 9**
    **A B C D F a b c d f**
    наприклад: **0x1F**

  можуть містити суфікс:
  - "**u**" або "**U**" для явного визначення без знакових констант (**unsigned**), наприклад: **2048u**
  - "**l**" або "**L**" для явного розширення типу константи до більшого (**long**),
    наприклад: **0xAFl**
  - "**ll**" або "**LL**"для явного подвійного розширення типу (**long long**),
    наприклад: **0xAFll**

  можуть містити одночасно комбінацію суфіксів:
  - "**u**" або "**U**" та "**l**" або "**L**", порядок немає значення,
    наприклад: **025ul**
  - "**u**" або "**U**" та "**ll**" або "**LL**", порядок немає значення,
    наприклад: **256ull**
- з плаваючою комою (*floating*)[2]:

---

[1] Крім десяткових, вісімкових та шістнадцяткових існує можливість декларувати двійкові константи за допомогою префіксу "**0b**" або "**0B**". Хоч ця можливість підтримується майже всіма компіляторами, однак вона не прописана в стандарті мови.

[2] В англомовному світі та в мові С зокрема плаваючою є крапка а не кома.

числа з плаваючою комою використовуються для наближення дробових та дійсних чисел. Вони містять значущу частину або мантису (*significant part*) після якої можуть слідувати екпоненціальна частина або порядок (*exponent part*) та суфікс, що явно визначає тип;

значуща частина зазвичай містить цілу та дробну частини, що записуються цифрами і розділені крапкою "**.**";

екпоненціальна частина починається одним з символів[1] "**e**", "**E**", після яких може слідувати знак "**+**" або "**-**" та обов'язково послідовність цифр;

для правильного запису повинна бути обов'язково наявна крапка, або екпоненціальна частина, наприклад:

```
1.5
02.
.025
1e-5
.8e3
3.e+6
```

числа з плаваючою комою можуть бути шістнадцятковими. Для цього використовується один з префіксів "**0x**", або "**0X**", а експоненціальна частина починається одним з символів "**p**", "**P**". При цьому значуща частина (але не експоненціальна!) може містити шістнадцяткові символи (**A-F** або **a-f**);

наприклад: **0x1E.2p3**

за замовчуванням тип константи з плаваючою комою без суфіксу є розширеним (**double**);

для явного вказання скороченого типу (**float**) використовуються суфікси "**f**", або "**F**" наприклад: **.1e4f**

для явного вказання подвійного розширеного типу (**long double**) використовуються суфікси "**l**", або "**L**";

наприклад: **.1e4l**

числа з плаваючою комою завжди є наближеними, при компіляції вони трансформуються у власний внутрішній формат (що відповідає стандарту IEEE 754 [9]). Усі константи, що записані в однаковій початковій формі обов'язково трансформуються в однаковий внутрішній формат, наприклад: **1.23**, **1.230**, **123e-2**, **123e-02**, та **1.23L** мають різну початкову форму і тому

[1] Для представлення чисел з плаваючою комою дуже часто використовується науковий формат. Наприклад число $1{,}5\times10^{-5}$ записується як **1.5e-5**, або **1.5E-5**.

внутрішнє представлення у них може відрізнятися. Це необхідно враховувати в операціях порівняння, ці числа не будуть рівними;

- перечислення (*enumeration*):
  використовуються для позначення елементів перечислення;
  для їх оголошення використовується ідентифікатор;
  ідентифікатор оголошений як перечислення завжди має цілочисельний тип (**int**), наприклад:

```
enum colors {RED, YELLOW, GREEN, BLUE};
```

  тут у фігурних дужках ідентифікатори **RED**, **YELLOW**, **GREEN** та **BLUE** є константами перечислення, їх можна використовувати в коді програми для змістовного позначення змінних типу **colors**, але фактично це числа типу **int**.
- символьні (*character*):
  записуються у одинарних лапках, наприклад: **'a'**
  якщо у символьній константі присутні кілька символів, наприклад **'ab'**, то поведінка програми не визначається стандартом, а залежить від конкретного компілятора;
  у лапках може бути записаний будь-який символ базового алфавіту крім одинарної лапки "**'**", зворотної нахиленої риски "****" та символу переходу на новий рядок;
  символьними константами можуть бути керуючі послідовності (*escape-sequence*):

| послідовність | шістнадцяткове значення | призначення |
|---|---|---|
| **\'** | **0x27** | записати одинарні лапки "**'**" |
| **\"** | **0x22** | записати подвійні лапки "**"**" |
| **\?** | **0x3f** | записати знак запитання "**?**" |
| **** | **0x5c** | записати зворотну нахилену риску "****" |
| **\a** | **0x07** | сигнал динаміка |
| **\b** | **0x08** | backspace |
| **\f** | **0x0c** | розрив сторінки |
| **\n** | **0x0a** | новий рядок |
| **\r** | **0x0d** | повернення каретки |
| **\t** | **0x09** | табулювання |
| **\v** | **0x0b** | вертикальне табулювання |

  у якості символьної константи можна явно вказати вісімковий код, наприклад: **'\37'**
  або шістнадцятковий код, наприклад: **'\xAA'**
  за замовчуванням тип символьної константи без префіксу завжди є

без знаковим символьним (**unsigned char**);

для явного розширення типу можуть використовуватися префікси "**L**", "**u**" або "**U**", наприклад у випадку використання Unicode.

Типи і розміри даних мови С

У мові С дані поділяються на 2 групи:

- складні або структуровані.
- прості або скалярні:

  для скалярних даних існують такі базові цілочисельні типи даних:

  - **char** – найменший цілочисельний тип даних, що може вмістити в собі окремі символи базового алфавіту. Згідно стандарту вміщає числа в діапазоні хоча б **[-127,127]**. Зазвичай це один байт;
  - **int** – цілочисельний тип, що згідно стандарту вміщає числа в діапазоні хоча б **[-32767, +32767]**, але зазвичай в сучасних 32-х розрядних компіляторах займає чотири байти;

  для скалярних даних існують такі базові типи даних з плаваючою комою:

  - **float** – число з плаваючою комою одиничної точності;
  - **double** – число з плаваючою комою подвійної точності.

  робота з ними регламентується стандартом IEEE 754 [9]:

| тип | байт | значущих біт | екпоненціальних біт | діапазон експоненти |
|---|---|---|---|---|
| **float** | **4** | **24** | **8** | **[-126, +127]** |
| **double** | **8** | **53** | **11** | **[-1022, +1023]** |

  - для розширення базових типів використовуються кваліфікатори **short** та **long**;

| Допустимі варіанти запису типу | мінімальний діапазон згідно стандарту | зазвичай байт |
|---|---|---|
| **short**<br>**short int** | **[-32767, +32767]** | **2** |
| **long**<br>**long int** | **[-2147483647L, +2147483647L]** | **4** |
| **long long**<br>**long long int** | **[-9223372036854775807LL, +9223372036854775807LL]** | **8** |

  кваліфікатор **long** може розширювати тип **double**, однак це розширення не регламентується стандартом IEEE 754, тому діапазони значень і розміри типу залежать від компілятора;

  для розширення базових типів використовуються кваліфікатори:

  - **signed** – явне вказання знакового типу;

- **unsigned** – явне вказання без знакового типу.

наприклад:

```
signed char
unsigned char
signed short
unsigned short int
unsigned long long
```

як випливає з означень, розмір базових типів і їх фактичні діапазони не регламентуються стандартом і залежать від конкретного компілятора, тому для визначення розміру базового типу завжди необхідно використовувати стандартну функцію часу компіляції **sizeof()**, а за необхідності визначення діапазонів конкретних типів – бібліотеку **<limits.h>**;

- довгий час у мові С не було логічного або булевського типу, хоча логічні операції використовувалися. Треба було запам'ятати, що значенню "істина" відповідає "не нуль", тобто будь-яке число, що не дорівнює нулю, а "не істина" – "нуль". З часом ситуація змінилася і в новому стандарті з'явився базовий логічний тип **_Bool**. Додатково в бібліотеці **<stdbool.h>** визначений більш звичний тип **bool**.
- **void** (*порожнеча*) – спеціальний тип, який містить пусту множину значень. Це неповний тип об'єкту, що використовується для явного вказання відсутності будь-якого типу, тому його використання відрізняється від інших типів даних. Зокрема він використовуєтья:
  - у якості типу повернення функції, якщо функція не повинна повертати значення;
  - у якості списку формальних параметрів функції, якщо функція таких немає;
  - у якості складової узагальненого вказівника, який може вказувати на будь-який об'єкт в пам'яті.

  У старих версіях компіляторів мови С цей тип був відсутній (з'явився зі стандартом С89). Якщо необхідно було зазначити, що функція не повертає результату, або немає формальних параметрів тип та параметри просто опускали. При цьому функція неявно повертала тип **int**. Узагальнені вказівники приводилися до типу вказівника на **char**. Неможливо оголосити змінну типу **void** та використовувати її у виразах, однак можна оперувати з вказівниками типу **void**.

Декларації

Всі змінні в програмі С повинні бути оголошені (задекларовані) до того як будуть використані. Деякі декларації можуть бути неявними. Декларація вказує тип і містить список одної або кількох змінних (ідентифікаторів):

```
int i, k, ms[20];
char C, cl;
```

Одночасно з декларацією змінних можлива і їх ініціалізація[1] (задання початкового значення за допомогою оператора присвоювання "**=**"[2]):

```
char ls = '0';
int k, l, lk[20], i = 1;
float eps = 1.0e-5;
```

До будь-якої змінної в декларації можна використати кваліфікатор **const**, тоді змінна перетворюється в константу, тобто її значення при виконанні програми не повинно змінюватися:

```
const int n = 20;
const double l = 2.71828182;
```

### 2.2.2.Структура програми

Програма на мові С умовно складається з двох частин:

- заголовка,
- тіла програми.

**Заголовок** складається з директив препроцесора і заголовків функцій. **Тіло програми** або функції являє собою набір операторів і міститься в фігурних дужках "**{}**". Ознакою закінчення оператора є символ крапка з комою "**;**". В одному рядку може бути кілька операторів, але бажано дотримуватись правила "одна стрічка – один оператор": програма більш наглядна і легше читається.

Коментарі в мові С обмежуються символами "**/***" і "***/**". Є можливість коментувати окремо рядки програми з допомогою символів "**//**".

Наведемо приклад простенької програми за допомогою якої вводимо з

---

[1] Згідно стандарту, явно непроініціалізовані змінні будуть містити невизначені значення, окрім випадку, якщо це статичні змінні.

[2] Оператор присвоювання "**=**" необхідний для збереження будь-яких значень в комірках пам'яті. Він є бінарним оператором, перший аргумент якого є *l-заченням* (*left*), *що модифікується* (*modifiable l-value*), тобто ідентифікатором, що посилається на деякий адрес в пам'яті – змінна, елемент масиву, поле структури даних, вказівник, адреса будь-якого з цих об'єктів. Другим аргументом є *r-значення* (*right, r-value*), тобто деякий об'єкт, що може бути присвоєний *l-заченню*, але не обов'язково є *l-заченням* – константа, вираз, результат виклику функції тощо. Наприклад, вираз **int a = 5**; тут **a** це *l-зачення*, що модифікується, а **5** – *r-значення*. Тому, вираз **5 = a;** є неправильним і при спробі його компіляції компілятор видасть повідомлення про помилку. В стандарті С11 терміни "*l-зачення, що модифікується*" та "*r-значення*" були замінені на "*значення локатора об'єкту*" та "*значення виразу*" відповідно.

клавіатури символ, а на екран виводимо код цього символу:

```
# include <stdio.h> /* header */
int main()
{
    char ch;
    printf("Enter symbol\n"); // body
    scanf("%c", &ch);
    printf("\nSymbol code %c:%d\n",ch,ch);
    return 0;
}
```

Кожна програма на мові С складається з функцій. В наведеному прикладі це функція **main()** – основна програма. Пусті дужки при імені функції означають, що ніяких вхідних параметрів основна програма не потребує. Стрічка **#include <stdio.h>** вказує компілятору, що необхідно включити інформацію, яка міститься у файлі **stdio.h** – стандартній бібліотеці вводу-виводу. Значок **#** означає, що це директива препроцесора, тобто такі команди повинні обробитися тільки один раз до процесу компіляції. Значки "**<>**" означають, що це стандартні файли, якими комплектується компілятор С.

В наведеному прикладі функція **main** складається з 4 операторів.

**char ch;** – оголошення змінної **ch** (ідентифікатор), типу **char.**

**printf("Enter symbol\n");** – виклик бібліотечної функції виводу на екран. В дужках задається список виводу. В даному випадку виводиться стрічка символів, що міститься в подвійних лапках: **Enter symbol**

**\n** – керуюча послідовність, що викликає перехід на наступну стрічку.

**scanf("%c", &ch);** – виклик бібліотечної функції форматного вводу.

Аргументами цієї функції є:

1) специфікатор формату: **%c** ;

2) вказівник на змінну **ch** – **&ch.**

Слід пам'ятати, що для того щоб ввести за допомогою функції **scanf** якесь значення і присвоїти його змінній одного з основних типів, перед іменем змінної необхідно записати символ "**&**".

Специфікатор формату відображає тип змінної, що виводиться на друк або вводиться з клавіатури. Розрізняють такі специфікатори формату (детельніше у лабораторній роботі №3):

**%d** – десяткове ціле число:

**%f** – число з плаваючою комою, десятковий запис;

**%e** – число з плаваючою комою, експоненціальний запис;

**%g** – число з плаваючою комою або десятковий або експоненціальний запис. Використовується тільки при виводі змінних;

**%c** – один символ;

**%s** – стрічка символів;

**%u** – десяткове ціле без знаку;

**%o** – вісімкове ціле число без знаку;

**%x** – шістнадцяткове ціле число без знаку;

допустиме використання префіксів для розширення чи звуження типів, наприклад **%lf** – для **double**.

Четвертий оператор тіла програми **printf("\nSymbol code %c:%d\n",ch,ch);** виведе на екран повідомлення "**Symbol code**", символ, який був введений з клавіатури, а тоді ціле число, що є кодом ASCII цього символа.

Операції

Умовно операції в мові С можна розбити на такі групи:

1) Арифметичні операції:
   a. унарні: **+;-;**
   b. бінарні **+;-;*;/**; (додавання, віднімання, множення, ділення).

   До операції ділення в мові С потрібно відноситись дуже уважно. Якщо обидва операнди цілого типу, то і результат буде цілого типу. Наприклад **S=2/5;** – в результаті виконання цього оператора, змінній **S** присвоїться значення **0**, щоб одержати правильний результат необхідно щоб хоча б один операнд був дійсного типу, тобто **S = 2.0/5.0**;

   У мові С є ще одна бінарна операція **%** - знаходження залишку від ділення цілих чисел. Наприклад **K=7%2;** – присвоїти змінній цілого типу **K** значення **1**, оскільки **7:2=3** і **1** в залишку. До змінних дійсного типу ця операція не застосовується.
2) Операції порівняння: **>**; **>=**; **<**; **<=**;
3) Операційні рівності: **==** – рівне; **!=** – не рівне
4) Логічні операції: **!** – логічне "ні"; **||** – логічне "або"; **&&** – логічне "і".
5) Інкрементні та декрементні операції.

   Інкрементна операція **++** додає 1 до свого операнда.

   Оператор **n++;** можна записати **n=n+1**;

   Декриментна операція **--** віднімає 1 від свого операнда.

   Розрізняють два види цих операцій:
   a. префіксні **++n** – змінна n збільшується на 1 до того, як використовується у виразі;
   b. постфіксні **n++** – змінна n збільшується на 1 після того, як її значення буде використано у виразі.

Для ілюстрації цих операцій виконаємо таку програму:

```
#include <stdio.h>
int main()
{   int a = 1, b = 1, aplus, plusb;
    aplus = a++;
    plusb = ++b;
    printf("  a aplus  b  plusb\n");
    printf("%3d%6d%3d%7d\n", a, aplus, b, plusb);
    return 0;
}
```

В результаті виконання цієї програми одержимо:

```
a aplus b plusb
2     1 2     2
```

Значення **a** збільшилось на 1 після того як виконалась операція присвоєння. Значення **b** спочатку збільшилось на 1, а тоді виконалась операція присвоєння.

6) Побітові операції

В мові С існує 6 операцій для роботи з бітами:

**&** – побітове “і”;

**|** – побітове “або”;

**^** – побітове “виключне “або” (XOR);

**~** – побітове “ні”;

**>>** – зсув вправо;

**<<** – зсув вліво.

**&** – *побітове “і”*. бінарна операція, що по розрядах порівнює два двійкові числа. Результат дорівнює 1, якщо обидва операнди рівні 1 у цьому розряді, тобто:

```
    1 0 0 1 0 0 1 1
&   0 0 1 1 1 1 0 1
  -----------------
    0 0 0 1 0 0 0 1
```

**|** – *побітове “або”***.** Результат 1, якщо хоча б у одного операнда у цьому розряді 1, тобто:

```
    1 0 0 1 0 0 1 1
|   0 0 1 1 1 1 0 1
  -----------------
    1 0 1 1 1 1 1 1
```

**^** – *побітове “виключне “або”*. Для кожного розряду результат дорівнює 1, якщо один з двох відповідних розрядів дорівнює 1, але не обидва одночасно:

```
    1 0 0 1 0 0 1 1
^   0 0 1 1 1 1 0 1
  -----------------
    1 0 1 0 1 1 1 0
```

**~** – *побітове "ні"*. Унарна операція, яка заміняє кожну 1 на 0, а 0 на 1:

```
~0b10010011 == 0b01101100
```

**>>** – *зсув вправо*. Зсуває розряди лівого, операнда вправо на кількість позицій вказаних у правому операнді:

```
0b10010011 << 2 == 0b00100100
```

**<< - зсув вліво** - зсуває розряди лівого операнда вліво на кількість розрядів, що вказані в правому операнді:

```
0b10010011 >> 2 == 0b01001100
```

позиції, які звільняються заповнюються нулями.

7) Тринарний оператор

Умовний оператор **if** у мові С можна замінити операцією виду "**?**"**:**"

```
z=(a<b)?a:b;
```

Цей оператор відповідає оператору умовного переходу такого виду:

```
if(a<b)
    z=a;
else
    z=b;
```

8) Операція присвоєння.

Операція присвоювання може мати такий вигляд:

```
<змінна> = <вираз>;
<змінна> <знак операції>= <вираз>;
```

наприклад:

```
S=S+4;
S+=4;
```

В операції присвоювання можуть використовуватись такі операції:

**+,-,*,/,%,<<,>>,&,|**

Пріоритет і порядок виконання операцій

| Пріоритет | Операції | Позначення | Порядок виконання |
|---|---|---|---|
| 1 | Виклик функції або вибір | `() [] -> .` | зліва-направо |
| 2 | Унарні операції | `+ -- ! & * ~` | справа-наліво |
| 3 | Мультиплікативні | `* / %` | зліва-направо |
| 4 | Адддитивні | `+ -` | зліва-направо |
| 5 | Зсуву | `>> <<` | зліва-направо |
| 6 | Порівняння | `> >= < <=` | зліва-направо |
| 7 | Рівності | `== !=` | зліва-направо |
| 8 | Побітове "і" | `&` | зліва-направо |
| 9 | Побітове виключне "або" | `^` | зліва-направо |
| 10 | Побітове "або" | `\|` | зліва-направо |
| 11 | Логічне "і" | `&&` | зліва-направо |

| | | | |
|---|---|---|---|
| 12 | Логічне “або” | `||` | зліва-направо |
| 13 | Умови | `?:` | справа-наліво |
| 14 | Присвоювання | `= <знак>=` | справа-наліво |
| 15 | Кома | `,` | зліва-направо |

Стандартні математичні функції бібліотеки **<math.h>**

| Ім’я функції | Математичний запис | Тип і межі зміни аргументів | Тип результату |
|---|---|---|---|
| **sin(x)** | sin x | **double** | **double** |
| **cos(x)** | cos x | **double** | **double** |
| **tan(x)** | tg x | **double** | **double** |
| **asin(x)** | arcsin x | **double** $x\in[-1,1]$ | $[-\pi/2,\pi/2]$ |
| **acos(x)** | arccos x | **double** $x\in[-1,1]$ | $[0,\pi]$ |
| **atan(x)** | arctg x | **double** $x\in[-\pi/2,\pi/2]$ | **double** |
| **sinh(x)** | sh x | **double** | **double** |
| **cosh(x)** | ch x | **double** | **double** |
| **tanh(x)** | th x | **double** | **double** |
| **exp(x)** | $e^x$ | **double** | **double** |
| **log(x)** | ln x | $x>0$ | **double** |
| **log10(x)** | lg x | $x>0$ | **double** |
| **pow(x,y)** | $x^y$ | **double** | **double** |
| **sqrt(x)** | $\sqrt{x}$ | $x\geq 0$ | **double** |
| **fabs(x)** | $\lvert x\rvert$ | **double** | **double** |
| **ldexp(x,n)** | $x \cdot 2^n$ | x-**double**, n-**int** | **double** |
| **fmod(x,y)** | Залишок від ділення дійсних чисел x на y | **double** | **double** |

2.2.3.Перетворення типів

В операторах і виразах бажано використовувати змінні і константи однакового типу. Якщо у виразі є змішування типів компілятор автоматично перетворить типи за такими правилами. Якщо операція виконується над змінними різних типів, то обидві змінні переводяться до “вищого” з двох типів.

Порядок типів від “вищого” до “нижчого” має такий вигляд:

1) **double**;
2) **float**;
3) **long**;
д) **int**;
5) **short**;
6) **char**.

В операторах присвоювання результат перетворюється до типу змінної, якій присвоюється це значення. Це може бути як “підвищення” типу так і “пониження”, наприклад в програмі:

```
#include<stdio.h>
int main()
{
    int n, m;
    float s, p;
    n = 5 / 7.0;
    m = 4.8 / 9.2;
    s = 5 / 7;
    p = 4.8 / 9.2;
    printf("  n   m    s      p \n");
    printf("%3d %3d  %3.1f  %5.3f\n",n,m,s,p);
    return 0;
}
```

одержимо результат:

```
  n    m    s      p
  0    0  0.0  0.521
```

Найкраще при написанні програми уникати перетворення типів, особливо в бік “пониження”. Можливий варіант явного приведення типів

**(<ім’я типу>)<змінна або константа>.**

Так у прикладі:

```
m = 1.6 + 1.8;
n = (int)1.6 + (int)1.8;
```

змінній **m** присвоїться значення **3**, оскільки спочатку додадуться два дійсних числа **1.6** і **1.8** від результату **3.4** відкинеться дробова частина. Змінній **n** присвоїться значення **2**, оскільки **(int)1.6==1**, **(int)1.8==1**; **1+1==2**.

### 2.2.4.Оператори

Будь-яка програма складається з послідовності операторів. Ознакою закінчення оператора є крапка з комою “**;**”. Так запис **S=5** не є оператором, це просто вираз, а **S=5;** це вже оператор присвоювання.

Блок – це група операторів, що міститься у фігурних дужках, вони використовуються:

- щоб згрупувати кілька логічно зв’язаних операторів в один;
- як тіло функції;
- для локалізації дії описів (визначають область видимості ідентифікаторів).

Оператор **if**

Оператор **if-else** використовується для вибору одного з двох варіантів рішення. Синтаксичний опис оператора **if-else**:

```
if(<вираз>)
    <оператор 1>;
else
    <оператор 2>;
```

Обчислюється "**вираз**", якщо його значення "істина" (тобто не нуль) виконується "**оператор 1**", якщо "не істина" (тобто нуль) виконується "**оператор 2**". Частина **else** може бути відсутня. При вкладених **if-else** необхідно пам'ятати, що **else** відноситься до внутрішнього **if**:

```
if(x>0)
    if(a>b)
        z = a;
    else
        z = b;
```

Якщо треба змінити порядок необхідно використати фігурні дужки, тобто виділити блок:

```
if(x>0)
{
    if(a>b) z = a;
}
else z = b;
```

Щоб розгалузити програму можна використати конструкцію: **else if**:

```
if(<вираз 1>)
    <оператор 1>;
else if(<вираз 2>)
    <оператор 2>;
else if(<вираз 3>)
    <оператор 3>;
else
    <оператор 4>;
```

Якщо "**вираз 1**" – "істина" виконується "**оператор 1**", якщо ні – перевіряється "**вираз 2**". Якщо "**вираз 2**" – "істина", виконується "**оператор 2**" і так далі.

Оператор **switch**

Оператор **switch** (перемикач) використовується для вибору одного з багатьох варіантів.

Синтаксис оператора **switch**:

```
switch(<вираз>)
{
case <константа вибору 1>:<оператор 1>;
case <константа вибору 2>:<оператор 2>;
…
default: <оператор n>
}
```

Якщо “вираз” співпадає з одною із констант вибору, то виконується відповідний оператор або блок операторів. Якщо “вираз” не співпадає з жодною з констант вибору – виконується оператор після слова “**default**”. Для прикладу промоделюємо роботу світлофора:

```
#include<stdio.h>
int main()
{
    char ch;
    printf("Enter first letter of color\n");
    scanf("%c",&ch);
    switch(ch)
    {
    case 'r': printf("Wait!\n"); break;
    case 'y': printf("Attention\n"); break;
    case 'g': printf("Go!\n"); break;
    default : printf("There is no such color!\n");
              break;
    }
    return 0;
}
```

“Вираз” і константи вибору повинні бути цілого типу, або типу **char.** Заборонено використовувати в якості константи вибору змінну. Оператор **break** здійснює негайний вихід з оператора **switch**. Якщо цього оператора немає, то будуть виконані оператори всіх варіантів після вибраного.

## 2.3. Контрольні запитання

1. Правила запису ідентифікаторів.
2. Які Ви знаєте типи даних і як вони описуються?
3. Структура програми на мові С.
4. Що таке побітові операції?
5. Що таке оператор, ознака закінчення оператора.
6. Як здійснюється узгодження типів у виразах?
7. Яка різниця між оператором і блоком у мові С?
8. Синтаксис умовного оператора **`if`**.
9. Напишіть приклад оператора **`switch`**.
10. Для чого використовується оператор **`break`**?

## 2.4. Лабораторне завдання

1. Вивчити основні поняття мови програмування С, операції, стандартні функції, оператори розгалуження програм.
2. Одержати індивідуальне завдання.
3. Побудувати блок-схеми алгоритмів відповідно до завдання.
4. Скласти програми на алгоритмічній мові С згідно завдання.
5. Відлагодити програми, виконати обчислення, проаналізувати отримані результати.

## 2.5. Зміст звіту

1. Титульний аркуш.
2. Мета роботи.
3. Індивідуальне завдання.
4. Блок-схема алгоритмів у відповідності до завдання.
5. Тексти програм у відповідності до завдання.
6. Результати обчислень.
7. Аналіз результатів, висновки.

## 2.6. Індивідуальні завдання

Завдання 1

Скласти програму обчислення функцій для різних значень її аргументів. Аргументи вводити з клавіатури. Вивести на екран значення функції. Передбачити у програмі обхід алгебраїчних операцій, які можуть при певних значеннях аргументів мати невизначений результат, тобто ділення на нуль, добування кореня парного степеня з від'ємного числа, логарифма від'ємного числа і тому подібне, при цьому вивести на екран повідомлення про те, що функція не визначена.

| № | Функція | Дані для перевірки |
|---|---|---|
| 1. | $t = \dfrac{2\cos(x-\pi/6)}{0{,}5+\sin^2 y}\left(1+\dfrac{z^2}{3-z^2/5}\right).$ | 0.5648 при<br>$x$ = 14.26,<br>$y$ = –1.22,<br>$z = 3.5\times10^{-2}$. |
| 2. | $u = \dfrac{\sqrt[3]{8+\lvert x-y\rvert^2+1}}{x^2+y^2+2} - e^{\lvert x-y\rvert}(\mathrm{tg}^2 z+1)^x$ . | –55.6848 при<br>$x$ = –4.5,<br>$y = 0.75\times10^{-4}$,<br>$z = 0.845\times10^{2}$. |
| 3. | $v = \dfrac{1+\sin^2(x+y)}{\left\lvert x-\dfrac{2y}{1+x^2y^2}\right\rvert} x^{\lvert y\rvert} + \cos^2(\mathrm{arctg}(1/z))$ . | 1.0553 при<br>$x = 3.74\times10^{-2}$,<br>$y$ = –0.825,<br>$z = 0.16\times10^{2}$. |
| 4. | $w = \lvert\cos x-\cos y\rvert^{\frac{1+2\sin^2 y}{x}}\left(1+z+\dfrac{z^2}{2}+\dfrac{z^3}{3}+\dfrac{z^4}{4}\right).$ | 0.9996 при<br>$x = 0.4\times10^{4}$,<br>$y$ = –0.875,<br>$z = -0.475\times10^{-3}$. |
| 5. | $\alpha = \ln(y^{-\sqrt{\lvert x\rvert}})(x-y/2)+\sin^2 \mathrm{arctg}(z).$ | –184.0359 при<br>$x$ = –15.246,<br>$y = 4.642\times10^{-2}$,<br>$z = 20.001\times10^{2}$. |
| 6. | $\beta = \sqrt{10(\sqrt[3]{x}+x^{y+2})}\cdot(\arcsin^2 z-\lvert x-y\rvert).$ | –40.6306 при<br>$x = 16.55\times10^{-3}$,<br>$y$ = –2.75,<br>$z$ = 0.15. |
| 7. | $\gamma = 5\mathrm{arctg}(x)-\dfrac{1}{4}\arccos(x)\dfrac{x+3\lvert x-y\rvert+x^2}{\lvert x-y\rvert z+x^2}.$ | 205.3055 при<br>$x$ = 0.1722,<br>$y$ = 6.33,<br>$z = 3.25\times10^{-4}$. |

| | | |
|---|---|---|
| 8. | $\phi = \dfrac{e^{\lvert x-y\rvert}\lvert x-y\rvert^{x+y}}{\operatorname{arctg} x + \operatorname{arctg} z} + \sqrt[3]{x^6 + \ln^2 y}.$ | 39.3741 при<br>$x = -2.235\times10^{-2}$,<br>$y = 2.23$,<br>$z = 15.221$. |
| 9. | $\psi = \lvert x^{y/x} - \sqrt[3]{y/x}\rvert + (y-x)\dfrac{\cos y - z/(y-x)}{1+(y-x)^2}.$ | 1.2131 при<br>$x = 1.825\times10^{2}$,<br>$y = 18.225$,<br>$z = -3.298\times10^{-2}$. |
| 10. | $a = 2^{-x}\sqrt{x + \sqrt[4]{\lvert y\rvert}}\ \sqrt[3]{e^{x-1/\sin z}}.$ | 1.2618 при<br>$x = 3.981\times10^{-2}$,<br>$y = -1.625\times10^{3}$,<br>$z = 0.512$. |
| 11. | $b = y^{\sqrt[3]{\lvert x\rvert}} + \cos^3 y\,\dfrac{\lvert x-y\rvert\cdot\left(1+\dfrac{\sin^2 z}{\sqrt{x+y}}\right)}{e^{\lvert x-y\rvert} + x/2}.$ | 0.7121 при<br>$x = 6.251$,<br>$y = 0.827$,<br>$z = 25.001$. |
| 12. | $c = 2^{y^x} + (3^x)^y - \dfrac{y\cdot(\operatorname{arctg} z - \pi/6)}{\lvert x\rvert + \dfrac{1}{y^2+1}}.$ | 4.2514 при<br>$x = 3.251$,<br>$y = 0.325$,<br>$z = 0.466\times10^{-4}$. |
| 13. | $f = \dfrac{\sqrt[4]{y + \sqrt[3]{x-1}}}{\lvert x-y\rvert(\sin^2 z + \operatorname{tg} z)}.$ | 0.3306 при<br>$x = 17.421$,<br>$y = 10.365\times10^{-3}$,<br>$z = 0.828\times10^{5}$. |
| 14. | $g = \dfrac{y^{x+1}}{\sqrt[3]{\lvert y-2\rvert}+3} + \dfrac{x+y/2}{2\lvert x+y\rvert}(x+1)^{-1/\sin z}.$ | 82.8256 при<br>$x = 12.3\times10^{-1}$,<br>$y = 15.4$,<br>$z = 0.252\times10^{3}$. |
| 15. | $h = \dfrac{x^{y+1} + e^{y-1}}{1 + x\lvert y - \operatorname{tg} z\rvert}(1 + \lvert y-x\rvert) + \dfrac{\lvert y-x\rvert^2}{2} - \dfrac{\lvert y-x\rvert^3}{3}.$ | –0.4987 при<br>$x = 2.444$,<br>$y = 0.869\times10^{-2}$,<br>$z = -0.13\times10^{3}$. |
| 16. | $w = \sqrt[3]{x^6 + \ln^2 y} + \dfrac{e^{\lvert x-y\rvert}\lvert x-y\rvert^{x+y}}{\operatorname{arctg} x + \operatorname{arctg} z}.$ | 39.3741 при<br>$x = -2.235\times10^{-2}$,<br>$y = 2.23$,<br>$z = 15.221$. |

Завдання 2

Скласти програму обчислення функцій для різних значень її аргументів. Аргументи вводити з клавіатури. Вивести на екран значення функції та проміжної змінної.

| № | Функція | Умова |
|---|---|---|
| 1. | $y=\ln(1+x^{1/5})+\cos^2(x+1),$ | $x=\begin{cases} z^2; & z<1; \\ z+1; & z\ge 1. \end{cases}$ |
| 2. | $y=\dfrac{2x+\cos\sqrt{\lvert x\rvert}}{x^2+5},$ | $x=\begin{cases} 2+z; & z<1; \\ \sin^2 z; & z\ge 1. \end{cases}$ |
| 3. | $y=-\pi x+\cos^2 x^3+\sin^3 x^2,$ | $x=\begin{cases} z; & z<1; \\ \sqrt{z^3}; & z\ge 1. \end{cases}$ |
| 4. | $y=2\cos^3 x^2+\sin^2 x^3-x,$ | $x=\begin{cases} z^3+0,2; & z<1; \\ z+\ln z; & z\ge 1. \end{cases}$ |
| 5. | $y=x-\ln(x+2,5)+3(e^x-e^{-x}),$ | $x=\begin{cases} -z/3; & z<-1; \\ \lvert z\rvert; & z\ge -1. \end{cases}$ |
| 6. | $y=\dfrac{2}{3}\sin^2 x-\dfrac{3}{4}\cos^2 x,$ | $x=\begin{cases} z; & z<0; \\ \sin z; & z\ge 0. \end{cases}$ |
| 7. | $y=\sin^3(x+x^2+x^3),$ | $x=\begin{cases} z^2-z; & z<0; \\ z^3; & z\ge 0. \end{cases}$ |
| 8. | $y=\sin^2 x+\cos^5 x^3+\ln x^{2/5},$ | $x=\begin{cases} 2z+1; & z\ge 0; \\ \ln(z^2-z); & z<0. \end{cases}$ |
| 9. | $y=\dfrac{x}{\cos x}+\ln\left\lvert \operatorname{tg}\dfrac{x}{2}\right\rvert,$ | $x=\begin{cases} z^2/2; & z\le 0; \\ \sqrt{z}; & z>0. \end{cases}$ |
| 10. | $y=\dfrac{xe^{\sin^3 x}+\ln(x+1)}{\sqrt{x}},$ | $x=\begin{cases} z^2+1; & z<1; \\ z-1; & z\ge 1; \end{cases}$ |
| 11. | $y=\dfrac{2,5e^{-3x}-4x^2}{\ln\lvert x\rvert+x},$ | $x=\begin{cases} \dfrac{1}{z^2+2z}; & z>0; \\ 1-z^3; & z\le 0. \end{cases}$ |
| 12. | $y=\sin^3(x^2-1)+\ln\lvert x\rvert+e^x,$ | $x=\begin{cases} z^2+1; & z\le 1; \\ 1/\sqrt{z-1}; & z>1. \end{cases}$ |

| | | |
|---|---|---|
| 13. | $y = \sin 3x + \cos x + \ln x$ | $x = \begin{cases} z; & z > 1; \\ z^2 + 1; & z \le 1. \end{cases}$ |
| 14. | $y = \cos 2x + \sin \frac{x}{5} + e^x,$ | $x = \begin{cases} \sqrt{z}; & z > 0; \\ 3z + 1; & z \le 0. \end{cases}$ |
| 15. | $y = x(\sin x + e^{-(x+3)}),$ | $x = \begin{cases} -3z; & z > 0; \\ z^2; & z \le 0. \end{cases}$ |
| 16. | $y = \ln \lvert x \rvert + e^x + \sin^3(x^2 - 1),$ | $x = \begin{cases} z^2 + 1; & z \le 1; \\ 1/\sqrt{z-1}; & z > 1. \end{cases}$ |

# ЛАБОРАТОРНА РОБОТА № 3.
# Оператори циклу, директиви препроцесора та форматований ввід-вивід у мові С

## 3.1. Мета роботи

Ознайомитися з директивами препроцесора мови С, з операторами циклу і функціями вводу-виводу.

## 3.2. Теоретичні відомості

### 3.2.1. Директиви препроцесора

Препроцесор мови С використовується для обробки тексту програми до її компіляції. Препроцесор виконує макропідстановку, умовну компіляцію, під'єднання іменованих файлів. Директиви препроцесора починаються з символу "**#**".

За допомогою директиви препроцесора **#include** в програму на мові С можна включити текст будь-якого файлу. Директива **#include** має дві форми:
**#include <ім'я файлу>** – під'єднання стандартного файлу:

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

**#include "<ім'я файлу>"** – під'єднання зовнішнього не стандартного файлу:

```
#include "myfile.h".
```

Суфікс "**h**" використовується для файлів, які під'єднуються в заголовку програми. Це, так звані, заголовочні (*header*) файли.

Макровизначення

Директива **#define** ставить у відповідність ідентифікатору текстову стрічку, тобто проводить деяке визначення. Синтаксис оператора:

**#define <ідентифікатор> <стрічка заміни>**

Стрічка заміни може містити ідентифікатори, ключові слова, розділювачі. Директива **#define** може стояти у будь-якому місці програми і виконує такі функції:

1) Визначення констант:

```
#define NULL 0
#define TRUE 1
#define FALSE 0
```

2) Прості макровизначення:

```
#define begin {
#define end }
```

тоді замість фігурних дужок будуть використовуватись слова **begin** і **end**.

3) Параметризація макровизначень:

```
#define MAX(x,y) ((x)>(y)?(x):(y))
#define MIN(x,y) ((x)>(y)?(y):(x))
```

Директива **#undef** – відміняє дію **#define**. Наприклад:

```
#include <stdio.h>
#define TWO 2
#define FOUR TWO*TWO
#define PX printf("x equal %d.\n",x)
int main()
{
    int x = TWO;
    PX;
    x = FOUR;
    PX;
    return 0;
}
```

У результаті роботи цієї програми одержимо повідомлення:

```
x equal 2.
x equal 4.
```

За директивою **#define** препроцесор замінює кожне макровизначення на стрічку заміни, тобто:

**int x = TWO** перетвориться в **int x = 2**;

**PX** перетвориться в **printf("x equal %d.\n",x)**

**x = FOUR** перетвориться в **x = TWO*TWO** і далі в **x = 2*2**

і так далі.

Умовна компіляція

Умовна компіляція це вибіркова компіляція лише тих частин програми, які задовольняють певні умови. Для умовної компіляції використовуються такі директиви препроцесора: **#if**, **#else**, **#endif**, **#ifdef**, **#ifndef**.

Синтаксис директиви умовної компіляції:

```
#if
<текстові рядки для випадку "істина">
#else
<текстові рядки для випадку "не істина">
#endif
```

**if** – заголовок містить умови, на основі яких здійснюється перевірка. Управляючий рядок **if** – заголовок має 3 форми:

```
#if <вираз, що має постійне значення>
#ifdef <ідентифікатор>
#ifndef <ідентифікатор>
```

В першій формі вираз визначається значенням нуль або не нуль (“істина”, “не істина”. В другій формі значення “істина” відповідає умові, якщо ідентифікатор був визначений в директиві **#define**. В третій формі значення “істина” відповідає умові, якщо ідентифікатор або не був визначений в директивою **#define**, або був відмінений директивою **#undef**.

Для прикладу умовної компіляції приведемо такий фрагмент програми:

```
#ifndef MAX_STK
# define MAX_STK 128
#endif
```

Ідентифікатор **MAX_STK** має значення по замовчуванню, якщо не буде заданий користувачем.

Директива **if** подібна до оператора **if** у мові С:

```
#if SYS == "IBM"
# include "ibm.h"
#endif
```

Якщо вираз **SYS=="IBM"** істина, то під’єднується файл “**ibm.h**”.

### 3.2.2.Оператори циклу у мові С

У мові С, як і в більшості інших існує три типи операторів циклу:

1) Оператор циклу з передумовою:

```
while(<вираз>) <оператор>;
```

2) Оператор циклу з постумовою:

```
do <оператор>;
while(<вираз>);
```

3) Оператор з параметрами:

```
for(<вираз 1>; <вираз 2>; <вираз 3>) <оператор>;
```

Наприклад, треба обчислити 5!. Фрагменти програм з операторами циклу будуть мати такий вигляд:

З оператором **while**:

```
int f = 1, n = 1;
while(n <= 5)
{
    f *= n;
    n++;
}
```

З оператором **do-while**:

```
int n = 1, f = 1;
do
{
   f *= n;
```

```
        n++;
    }
    while(n <= 5);
```

З оператором **for**:

```
    int f, n;
    for(f = 1, n = 1; n <=5 ; n++)
        f *= n;
```

Оператори циклу **while** виконуються до того часу поки виконується умова, тобто **n<=5**. Якщо умова не виконується, наприклад **n==8**, то оператори циклу **while** не виконуються ні разу.

Оператори циклу **do-while** також виконуються до того часу поки виконується умова. Але перевірка умови проводиться після першого виконання циклу, тобто якщо умова одразу не виконується, наприклад **n==8**, то оператори циклу **do-while** один раз будуть виконані.

Оператор циклу **for** можна подати в такому вигляді:

```
    for(<ініціалізація початкових значень>;
      <перевірка умови>;
      <зміна параметра>)
    <оператор>;
```

У прикладі надаємо початкові значення не тільки параметру циклу **n**, але і змінній **f**. Далі перевіряється умова виконання циклу **n<=5**, якщо умова виконується[1], то виконуються оператори циклу. Третій вираз це зміна параметра циклу. У нашому випадку **n=n+1** або **n++**. Оператор **for** має дуже гнучку структуру. Він може мати вкорочену форму, тобто:

```
    for(;n <= 5;)
    f *= n;
```

але тоді зміну **n** треба робити в тілі операторів циклу, а визначення початкового значення перед оператором **for**.

Допускається і така форма запису оператора **for**:

```
    int y = 1;
    for(int x = 1; y <= 15; y = 5 * x++)
        printf("%10d%10d\n", x, y);
```

---

[1] Часто в циклі **for** можна зустріти використання оператора *кома* “**,**”. Цей оператор, згідно зі стандарту є однією з *точок слідування* – спеціальних мість, наприклад, таких як оператор “**;**”, що гарантують виконання всіх побічних ефектів (оголошення, ініціалізація, виклик функцій, обчислення виразів і т.д.) у виразі ліворуч. Особивістю оператора “**,**” є те, що він відкидає результати виразу ліворуч і повертає у місце виклику результат виразу праворуч. Наприклад у конструкції **for(;a < 5, b > 10;){/*...*/}** обидва логічні вирази будуть обчислені, але до уваги буде прийматися тільки останній результат **b > 10**.

В результаті роботи цієї програми одержимо:

```
1          1
2          5
3         10
4         15
```

Тут перевіряється умова виходу по значенню **y**, а не **x**, а в виразі “зміна параметра” одночасно рахується значення **y** і **x** змінюється на 1. В мові С допускається вкладення циклів. Вкладеним називається цикл, що міститься всередині іншого циклу. Для ілюстрації приведемо програму, яка буде виводити на друк всі прості числа, що містяться між числом **2** і **num**:

```
#include <stdio.h>
int main()
{   int number, div, num, count = 0;
    printf("please input integer number > 2\n");
    scanf("%d", &num);
    printf("prime numbers between 2 and %d are:\n",
        num);
    for(number = 2; number <= num; number++)
    {   for(div = 2; number % div != 0; div++);
            if(div == number)
            {   printf("%5d", number);
                if(++count % 10 == 0)
                    printf("\n");
            }
    }
    return 0;
}
```

Якщо ввести ціле число 100, то в результаті роботи програми одержимо прості числа в діапазоні від 2 до 100:

```
please input integer number > 2
100
prime numbers between 2 and 100 are:
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97
```

### 3.2.3. Управляючі оператори **break**, **continue** і **goto**

Оператор **break** здійснює негайний вихід з операторів циклу або оператора **swich**. Управління передається наступному оператору після оператора з якого здійснювався вихід. Якщо оператор **break** стоїть всередині

вкладеного циклу, то вихід здійснюється тільки із внутрішньої структури, тобто тільки з того циклу в якому є оператор **break**.

Оператор **continue** – передає управління на кінець тіла циклу, всередині якого він знаходиться. Тобто пропускає частину ітерації, яку виконує і переходить до наступної ітерації. Наприклад, треба знайти суму додатних чисел, що вводяться з клавіатури:

```
#include <stdio.h>
int main()
{
    int n, s = 0, x;
    printf("please input the number of elements:\n");
    scanf("%d",&n);
    printf("please input %d integer numbers:\n", n);
    for(int i = 0; i < n; i++)
    {
        scanf("%d",&x);
        if(x < 0) continue;
        s += x;
    }
    printf("the sum of positive numbers is %d\n", s);
    return 0;
}
```

Результати:

```
please input the number of elements:
5
please input 5 integer numbers:
1 2 -3 -4 5
the sum of positive numbers is 8
```

Оператор **goto**:

```
goto <мітка>
```

де: “мітка” – це мітка оператора на який здійснюється перехід. Міткою може бути будь-який ідентифікатор, після якого стоїть символ двокрапка “**:**”.

Мова С володіє такими засобами, що використовувати оператор **goto** немає потреби. Єдиний випадок коли можна використовувати оператор **goto** це вихід із внутрішнього, вбудованого циклу у випадку знаходження помилки:

```
for(i = 0; i < n; i++)
     for(j = 0; j < m; j++)
          if(a[i] == b[j]) goto err;
...
err: printf("Error!\n");
```

### 3.2.4.Специфіка використання операторів **break** і **continue**.

Оператор **break**, який стоїть в тілі циклу, негайно припиняє виконання циклу й передає керування на рівень вище, а точніше, на наступний оператор, що стоїть після даного циклу, який містить **break**. Тому для припинення виконання багаторівневого циклу по "ініціативі" на найглибшому рівні доводиться виконувати не один, а декілька операторів **break.** Наприклад треба знайти суму перших додатних чисел, що вводяться з клавіатури:

```
#include <stdio.h>
int main()
{
    int n, s = 0, x;
    printf("please input positive numbers:\n");
    while(1)
    {   scanf("%d",&x);
        if(x < 0) break;
        s += x;
    }
    printf("the sum of positive numbers is %d\n", s);
    return 0;
}
```

Результати:

```
please input positive numbers:
1 2 3 4 5 0 -2
the sum of positive numbers is 15
```

Ще один приклад використання оператора **continue**:

```
#include <stdio.h>
int main()
{
    char ch;
    while((ch = getchar())!= '\n')
    {   if(ch == '*') continue;
        putchar(ch);
    }
    return 0;
}
```

У даному прикладі у випадку вводу символу ***** керування програми передається знову на початок циклу. При цьому оператор **putchar(ch)** ігнорується. Якщо в цьому випадку використовувати оператор **break** замість **continue**, то ввід символу ***** аналогічний вводу символу переводу рядка, тобто виходу із циклу.

Функція **getchar()** читає символ з клавіатури, а функція **putchar()** – виводить його на екран.

### 3.2.5. Форматований ввід-вивід

Функції **printf()** і **scanf()** виконують форматований ввід-вивід на консоль, інакше кажучи, вони можуть зчитувати й записувати дані в заданому форматі, Функція **printf()** виводить дані на консоль. Функція **scanf()**, навпаки, зчитує дані з клавіатури. Обидві функції можуть оперувати будь-якими вбудованими типами даних, включаючи символи, рядки та числа.

Функція **printf()** – *print formatted*

Прототип функції **printf()** виглядає таким чином:

```
int printf (const char  *<керуюча_стрічка>,   ...);
```

Функція повертає кількість записаних нею символів, а у випадку помилки – від'ємне число. Параметр **<керуюча_стрічка>** складається з елементів двох видів. По-перше, він містить символи, які виводяться на екран. По-друге, у нього входять специфікатори формату, що починаються зі знака відсотка, за яким слідує код формату. Кількість аргументів повинна співпадати з кількістю специфікаторів формату, вони попарно зрівнюються зліва направо. Наприклад:

```
printf("I love %c%s", 'C', "11!");
```

виведе на екран рядок:

```
I love C11!
```

Функція **printf()** допускає широкий вибір специфікаторів формату, зокрема:

| Код | Формат |
|---|---|
| **%c** | Символ |
| **%d, %i** | Десяткове ціле число зі знаком |
| **%e, %E** | Науковий формат |
| **%f** | Десяткове число із плаваючою комою |
| **%g, %G** | Залежно від того, який формат коротший, застосовується або **%e**, або **%f** |
| **%a, %A** | Шістнадцяткове число з плаваючою комою |
| **%o** | Вісімкове число без знаку |
| **%s** | Рядок символів |
| **%u** | Десяткове ціле число без знаку |
| **%x, %X** | Шістнадцяткове число без знаку (малі літери) |
| **%p** | Вказівник |
| **%n** | Вказівник на цілочисельну змінну. Специфікатор викликає присвоювання цій цілочисельній змінній кількість символів, виведених перед ним |
| **%%** | Знак **%** |

Вивід символів

Для виводу окремих символів використовується специфікатор **%c**. У результаті відповідний аргумент без змін буде виведений на екран. Для виводу рядків застосовується специфікатор **%s**.

Вивід чисел

Для виводу десяткових цілих чисел зі знаком застосовуються специфікатори **%d** або **%i**. Ці специфікатори еквівалентні. Одночасна підтримка обох специфікаторів обумовлена історичними причинами. Для виводу цілого числа без знака варто застосовувати специфікатор **%u**. Специфікатор формату **%f** дає змогу виводити на екран числа із плаваючою комою. Специфікатори **%e** та **%E** вказують функції **printf()**, що на екран виводиться аргумент типу **float** у науковому форматі. Числа, представлені в науковому форматі, виглядають так:

```
x.dddddE+/-yy
```

Якщо буква **E** повинна бути виведена як велика, варто використовувати специфікатор **%E**, а якщо як мала – **%e**.

Функція **printf()** може сама вибирати подання числа за допомогою специфікатора **%f** або **%e**, якщо замість них вказати специфікатори **%g** або **%G**. У цьому випадку функція сама визначить, який вид числа коротший. Специфікатор **%G** дає змогу вивести букву **E** як велику, а **%g** – як малу. Наступна програма демонструє ефект застосування специфікатора **%g**:

```
#include <stdio.h>
int main()
{
    double f;
    for(f = 1.0; f < 1.0e+10; f *= 10)
        printf("%g ", f);
    return 0;
}
```

У результаті на екрані з'являться такі числа:

```
1 10 100 1000 10000 100000 1e+06 1e+07 1e+08 1e+09
```

Вивід адрес

Якщо на екран необхідно вивести адресу, варто застосовувати специфікатор **%p**. Цей специфікатор формату змушує функцію **printf()** виводити на екран адресу, формат якої сумісний з типом адресації, прийнятої в комп'ютері. Наступна програма виводить на екран адресу змінної **sample**:

```
#include <stdio.h>
int sample;
int main()
```

```
{
    printf("%p", &sample);
    return 0;
}
```

Можливий результат:

```
  0x600a24
```

Специфікатор **%n**

Специфікатор формату **%n** відрізняється від всіх інших. Він змушує функцію **printf()** записувати у відповідну змінну кількість символів, уже виведених на екран. Специфікатору **%n** повинен відповідати цілочисельний вказівник. Після завершення функції **printf()** цей вказівник буде посилатися на змінну, у якій утримується кількість символів, виведених до специфікатора **%n**. Наприклад:

```
#include <stdio.h>
int main()
{
    int count;
    printf("It is %n test\n", &count);
    printf("%d", count);
    return 0;
}
```

Ця програма виведе на екран рядок "**It is  test**" і число **6**. Специфікатор **%n** зазвичай використовується для динамічного форматування.

Модифікатори формату

Багато специфікаторів формату мають свої модифікатори, які трохи змінюють їхній зміст. Наприклад, з їхньою допомогою можна змінювати мінімальну ширину поля, кількість цифр після десяткової коми, а також виконувати вирівнювання по лівому краю. Модифікатор формату вказується між символом відсотка і кодом формату. Кожен модифікатор формату може доповнюватися відповідно до прототипу:

```
%[прапорець][ширина][.точність][розширення]специфікатор
```

Прапорець

| | |
|---|---|
| **–** | вирівнювання ліворуч (за замовчуванням праворуч); |
| **+** | примусово виводить знак числа, навіть якщо воно додатнє; |
| (пробіл) | якщо число додатнє, виведе пробіл, якщо від'ємне – знак мінус; |
| **#** | якщо використовується з **%o** та **%x** то примусово виводить перед числом **0** або **0x** відповідно. Якщо з **%a**, **%e**, **%f** чи **%g**, то примусово виводит символ ".", навіть якщо число ціле; |
| **0** | доповнює число ліворуч ведучими нулями. |

Модифікатор мінімальної ширини поля

Ціле число, розміщене між символом відсотка і кодом формату, задає мінімальну ширину поля. Якщо рядок виводу коротший, ніж потрібно, він доповнюється пробілами, якщо довший – рядок все рівно виводиться повністю. Наприклад:

```
#include <stdio.h>
int main()
{   float item = 10.12304;
    printf("%f\n", item);
    printf("%12f\n", item);
    printf("%012f\n", item);//leading zeros using flag
    return 0;
}
```

Ця програма виводить на екран наступні числа:

```
  10.123040
     10.123040
  00010.123040
```

Модифікатор мінімальної ширини поля найчастіше використовується для форматування таблиць. Програма, наведена нижче, створює таблицю квадратів і кубів чисел.

```
#include <stdio.h>
int main()
{   int i;
    for(i = 1; i <= 8; i++)
        printf("%8d %8d %8d\n", i, i*i, i*i*i);
    return 0;
}
```

У результаті на екран буде виведена наступна таблиця:

```
 1       1       1
 2       4       8
 3       9       27
 4       16      64
 5       25      125
 6       36      216
 7       49      343
```

Модифікатор точності

Модифікатор точності вказується після модифікатора ширини поля (якщо він є). Цей модифікатор складається із крапки, за якої слідує ціле число. Точний зміст модифікатора залежить від типу даних, до яких він застосовується.

Якщо модифікатор точності застосовується до чисел із плаваючою комою з

форматами **%f**, **%e** або **%E**, він означає кількість десяткових цифр після крапки. Наприклад, специфікатор формату **%10.4f** означає, що на екран буде виведене число, яке складається з дев'яти символів, чотири з яких розташовані після крапки, тобто, чотири цифри до крапки, сама крапка і до п'яти цифр перед крапкою.

Якщо модифікатор застосовується до специфікаторів формату **%g** або **%G**, він задає кількість значущих цифр.

Якщо модифікатор використовується для виводу рядків, він задає максимальну довжину поля. Наприклад, специфікатор **%5.7s** означає, що на екран буде виведений рядок, який складається як мінімум з п'яти символів, довжина якого не перевищує семи символів. Якщо рядок виявиться довшим, останні символи будуть відкинуті.

Якщо модифікатор точності застосовується до цілих типів, він задає мінімальну кількість цифр, з яких повинне складатися число. Якщо число складається з меншої кількості цифр, воно доповнюється ведучими нулями. Наприклад:

```
#include <stdio.h>
int main()
{
    printf("%.4f\n", 123.1234567);
    printf("%.8d\n", 1000);
    printf("%8.10s\n", "It is simple test.");
    return 0;
}
```

Ця програма виводить на екран наступні результати:

```
  123.1235
  00001000
  It is simp
```

Модифікатор розширення

| | специфікатор % | | |
|---|---|---|---|
| розширення | **d i** | **u o x** | **f e g a** |
| (немає) | **int** | **unsigned int** | **double** |
| **hh** | **signed char** | **unsigned char** | |
| **h** | **short** | **unsigned short** | |
| **l** | **long** | **unsigned long** | |
| **ll** | **long long** | **unsigned long long** | |
| **L** | | | **long double** |

Функція **scanf()** – *scan formatted*

Функція є процедурою вводу. Вона може зчитувати дані всіх вбудованих типів і автоматично перетворювати числа у відповідний внутрішній формат.

Дана функція є повною протилежністю до функції **printf()**. Прототип функції **scanf()** має такий вигляд:

```
int scanf (const char  *<керуюча_стрічка>,  ...);
```

Функція повертає кількість змінних, яким вона успішно присвоїла значення. Якщо при читанні відбулася помилка, функція **scanf()** повертає константу **EOF**. Параметр **<керуюча_стрічка>** визначає порядок зчитування значень і присвоювання їх змінним, зазначеним у списку аргументів.

Керуюча стрічка складається із символів, розділених на три категорії:

- специфікатори формату;
- розділювачі;
- символи, що не є розділювачами.

Ввід чисел

Для вводу цілого числа використовуються специфікатори **%d** або **%i**. Специфікатор **%i** підтримує ввід вісімкових (починаються з **0**) та шістнадцяткових (починаються з **0x**) чисел. Для вводу числа із плаваючою комою, представленого в стандартному або науковому форматі, застосовуються специфікатори **%e**, **%f** або **%g**, а також **%a**, для вводу шістнадцяткових чисел з плаваючою комою. Використовуючи специфікатори **%o** або **%x**, можна вводити цілі числа, представлені у вісімковому або шістнадцятковому форматі відповідно. Наприклад:

```
#include <stdio.h>
int main()
{
    int i, j;
    scanf("%o%x", &i, &j);
    printf("%o %x", i, j);
    return 0;
}
```

Функція **scanf()** припиняє вводити числа, виявивши перший нечисловий символ.

Ввід цілих чисел без знаку

Для вводу цілих чисел без знаку застосовується модифікатор **%u**. Наприклад, фрагмент:

```
unsigned num;
scanf("%u", &num);
```

вводить ціле число без знаку і присвоює його змінній **num**.

Ввід окремих символів

Як вказувалося раніше, окремі символи можна вводити за допомогою функції **getchar()** або похідних від її функцій. Функцію **scanf()** також

можна застосовувати для цієї мети, використовуючи специфікатор **%c**. Однак, як і функція **getchar()**, функція **scanf()** використовує буферизований ввід, тому в інтерактивних програмах її застосовувати не слід.

Незважаючи на те що пробіли, знаки табуляції й символи переходу на новий рядок використовуються як розділювачі при читанні даних будь-яких типів, при вводі окремих символів вони зчитуються нарівні з усіма. Наприклад, якщо потік вводу містить рядок "**x y**", то фрагмент коду:

```
scanf("%c%c%c", &a, &b, &c);
```

присвоїть символ "**x**" змінній **a**, "пробіл" – змінній **b** і символ "**y**" – змінній **c**.

Ввід рядків

Функцію **scanf()** можна застосовувати для вводу рядків із вхідного потоку. Для цього використовується специфікатор **%s**. Він змушує функцію **scanf()** зчитувати символи, поки не виявиться розділювач. Символи, пораховані із вхідного потоку, записуються в масив, на який посилається відповідний аргумент, а в кінець цього масиву записується нульовий байт. Функція **scanf()** вважає розділювачем "пробіл", символ переходу на новий рядок "**\n**", символ табуляції "**\t**", символ вертикальної табуляції "**\v**", а також символ розриве сторінки "**\f**". Отже, функцію **scanf()** не можна просто застосувати для вводу рядка "**It is test**", оскільки ввід припиниться на першому ж пробілі. Наприклад:

```
#include <stdio.h>
int main()
{   char str[80];
    printf("Please input a string:\n");
    scanf("%s", str);
    printf("This is your string: %s", str);
    return 0;
}
```

Результати:

```
Please input a string:
It is test
This is your string: It
```

Модифікатори формату

Як і у функції **printf()**, для **scanf()** кожен модифікатор формату може доповнюватися відповідно до прототипу:

```
%[*][ширина][розширення]специфікатор
```

***** вказує на те, що інформація має бути прочитана, але не повинна записуватися в змінні, що розташовані після керуючої стрічки;

**ширина** максимальна кількість символів, що повинні бути прочитані;

**розширення**:

| розширення | специфікатор % d i | u o x | f e g a |
|---|---|---|---|
| (немає) | int* | unsigned int* | float* |
| hh | signed char* | unsigned char* | |
| h | short* | unsigned short* | |
| l | long* | unsigned long* | double* |
| ll | long long* | unsigned long long* | |
| L | | | long double* |

Крім того, для функції **scanf()** визначено специфікатор **[<символи>]** та **[^<символи>]**, що позначають, які символи повинні читатися, чи які повинні розглядатися як термінатори при читанні відповідно. Тому, існує можливість обійти незручність функції **scanf()** для вводу рстрічок з розділювачами явно вказуючи які вхідні символи слід розгядати як кінець вводу, за допомогою конструкції **%[^<символ 1><символ 2>...]**. Наприклад: **scanf( "%[^\n]", str);** – зчитування продовжуватиметься, доки не зустрінеться символ "**\n**". Детальнішу інформацію про модифікатори функцій **printf()** та **scanf()** можна знайти у відповідній документації. Також часто використовуються аналоги цих функцій, що працюють не з стандартним вводом-виводом, а наприклад з файлами **fprintf()** та **fscanf()**, чи стрічками **sprintf()** та **sscanf()**.

3.2.6.Функції і перемикання вводу-виводу.

У мові С, наслідуючи концепцію операційної системи UNIX, що розглядає будь-яку перефирію як деякий файл (концепція – "*все довкола файл*"), стандартним вводу та виводу відповідають файли з зарезервованими назвами **stdin** та **stdout**, що за замовчуванням відповідають клавіатурі та екрану консолі. Відповідно, функції **scanf()** та **printf()** працюють з цими файлами. Крім описаних функцій в стандартній бібліотеці вводу-виводу **<stdio.h>** існують і інші. Зокрема функції вводу і виводу одного символа: **getchar()** та **putchar()**. Ці функції використовуються для вводу-виводу текстів. Функція **getchar()** одержує один символ з клавіатури і записує його в стандартний файл вводу **stdin**. Функція **putchar(char a)** пересилає один символ з пам'яті машини в стандартний файл виводу **stdout** (тобто на екран). Найпростіша програма, яка відображає роботу цих функцій має вигляд:

```
#include <stdio.h>
int main()
{   char ch;
    ch = getchar();
```

```
        putchar(ch);
        return 0;
    }
```

Параметром функції **putchar()** є ім'я змінної, що виводиться на друк. Функція **getchar()** параметрів не має.

Щоб відмітити де закінчується один файл і починається інший вводиться таке поняття, як "*ознака кінця файлу*" – **EOF** (*End-of-File*)[1]. З використанням символьної константи **EOF** програма копіювання із стандартного файлу вводу **stdin** в стандартний файл виводу **stdout** має вигляд:

```
#include <stdio.h>
int main()
{   char ch;
    while((ch=getchar())!=EOF) putchar(ch);
    return 0;
}
```

Якщо існує потреба вводити інформацію не з стандартного файлу (тобто з клавіатури), а з файлу що міститься на якомусь іншому периферійному пристрою, необхідно вказати комп'ютеру що джерело даних є файл, а не клавіатура. Це можна зробити двома методами:

1) Явно, використовуючи стандартні функції, що відкривають і закривають файли, організують зчитування і запис даних.
2) Не змінюючи програми (тобто функцій вводу-виводу), а використовуючи при її запуску засоби консольних команд операційної системи перемкнути ввід-вивід, тобто вказати комп'ютеру при виконанні програми, що вхідні дані містяться не у стандартному файлі **stdin**, а наприклад у файлі **data.txt**.

Історично операція перемикання вводу-виводу – це засіб операційної системи UNIX, а не самої мови С. Але вона виявилась настільки корисною, що при переносі компілятора з мови С на інші системи найчастіше переноситься і ця операція. Перемикання вводу здійснюється за допомогою знака "**<**". Наприклад, якщо програма **test.exe** для вводу використовує функцію **getchar()**, то командна стрічка

```
test.exe<data.txt
```

вказує програмі, що вхідні дані вводяться не з клавіатури, а з файлу **data.txt**.

Перемикання виводу здійснюється за допомогою знаку "**>**".

```
test.exe>rez.txt
```

Вивід результатів **test.exe** буде здійснюватись у файл **rez.txt**.

---

[1] Для лінійки операційних систем Windows, **EOF** на консолі відповідає комбінація клавіш **Ctrl+Z**.

### 3.3. Контрольні запитання

1. Що таке директиви препроцесора, для чого вони існують?
2. Які функції має директива **`#define`**?
3. Які директиви умовної компіляції?
4. Які Ви знаєте оператори циклу у мові С?
5. Що Ви знаєте про оператори **`break`** і **`continue`**?
6. Які функції вводу-виводу Ви знаєте?
7. Що таке перемикання вводу-виводу?

### 3.4. Лабораторне завдання

1. Ознайомитися з директивами препроцесора мови С, з операторами циклу і функціями вводу-виводу.
2. Одержати індивідуальне завдання.
3. Побудувати блок-схеми алгоритмів відповідно до завдання.
4. Скласти програми на алгоритмічній мові С згідно завдання.
5. Відлагодити програми, виконати обчислення, проаналізувати отримані результати.

### 3.5. Зміст звіту

1. Титульний аркуш.
2. Мета роботи.
3. Індивідуальне завдання.
4. Блок-схема алгоритмів у відповідності до завдання.
5. Тексти програм у відповідності до завдання.
6. Результати обчислень.
7. Аналіз результатів, висновки.

### 3.6. Індивідуальні завдання

Завдання 1

Скласти програму для обчислення значень функції для різних значень аргументу (протабулювати функцію) на вказаному відрізку, використовуючи три оператори циклу. Обчислити означений інтеграл функції на вказаному відрізку. Значення аргументу розглянути у вказаній кількості точок, задавши її як константу препроцесора. Вивести на екран по стовпчиках номер за порядком, значення аргументу і значення функції, використовуючи можливості форматованого виводу. Окремо вивести значення обчисленого інтегралу.

| № | Функція | Відрізок | К-сть. вузлів |
|---|---|---|---|
| 1. | $y = \arccos(\cos x),$ | $[-\pi/2, \pi/2],$ | N = 11. |
| 2. | $y = \arccos\frac{1-x^2}{1+x^2},$ | $[-0.5, 0.5],$ | N = 17. |
| 3. | $y = \log\frac{2-x}{1+x},$ | $[0.0, 1.0],$ | N = 13. |
| 4. | $y = \ln(x^2 + 2x - 1),$ | $[1.0, 2.0],$ | N = 17. |
| 5. | $y = \operatorname{arctg} x + \operatorname{arctg}\frac{1-x}{1+x},$ | $[-1.0, 1.0],$ | N = 11 |
| 6. | $y = e^{-x} + \sin(2x) - 1.5x^2,$ | $[0.0, 1.0],$ | N = 13. |
| 7. | $y = \frac{1}{\pi}\operatorname{arctg}\left(\frac{2x-1}{2} + \pi\right),$ | $[0, \pi],$ | N = 15. |
| 8. | $y = \lvert x^2 - 3x + 0.1\rvert - \frac{e^x}{x-1},$ | $[2.0, 3.0],$ | N = 15. |
| 9. | $y = \frac{x}{1-x^2} + \frac{\sin x}{\cos x},$ | $[0.0, 0.5],$ | N = 17. |
| 10. | $y = \log_2(x+3) + \ln(x+2),$ | $[1, 2],$ | N = 11. |
| 11. | $y = 2\sin 2x - 1.5x,$ | $[0, \pi],$ | N = 15. |
| 12. | $y = \sqrt{e^x} - \sin x,$ | $[0, \pi]$ | N = 13. |
| 13. | $y = \frac{1}{3 + 2\cos x},$ | $[1.0, 2.0],$ | N = 17. |
| 14. | $y = \sin\frac{x}{2} + 1.5\cos^2\frac{x}{3},$ | $[0, \pi],$ | N = 17. |
| 15. | $y = \sin 4x + 2x\cos(x - \pi),$ | $[0, \pi],$ | N = 13. |
| 16. | $y = 1 + x^2 - \sin 2x,$ | $[0, \pi/2],$ | N = 13. |

Завдання 2

Скласти програму для наближеного обчислення значення функції $Y(x)$ в точці $0 < |x| < 1$ за допомогою розкладу в ряд Тейлора $S(x)$. Знайти наближене значення функції з похибкою менше $\varepsilon < 0{,}0001$. Значення $x$ та $\varepsilon$ вводити з клавіатури. Вивести на екран точне значення $Y(x)$, знайдене наближене значення $S(x)$ та отриману похибку $|S(x) - Y(x)|$.

| № | Функція | Розклад в ряд Тейлора |
|---|---|---|
| 1. | $Y(x)=\sin(x),$ | $S(x)=\sum_{k=0}^{\infty}(-1)^k\frac{x^{2k+1}}{(2k+1)!}.$ |
| 2. | $Y(x)=x\cdot\operatorname{arctg}(x)-\ln\sqrt{1+x^2},$ | $S(x)=\sum_{k=1}^{\infty}(-1)^{k+1}\frac{x^{2k}}{2k(2k-1)}.$ |
| 3. | $Y(x)=\frac{1}{1-x},$ | $S(x)=1+\sum_{k=1}^{\infty}x^k.$ |
| 4. | $Y(x)=\cos(x),$ | $S(x)=\sum_{k=0}^{\infty}(-1)^k\frac{x^{2k}}{(2k)!}.$ |
| 5. | $Y(x)=e^{\cos x}\cos(\sin(x)),$ | $S(x)=\sum_{k=0}^{\infty}\frac{\cos(kx)}{k!}.$ |
| 6. | $Y(x)=(1+2x^2)e^{x^2},$ | $S(x)=\sum_{k=0}^{\infty}\frac{2k+1}{k!}x^{2k}.$ |
| 7. | $Y(x)=-\frac{1}{2}\ln(1-2x\cos\frac{\pi}{3}+x^2),$ | $S(x)=\sum_{k=1}^{\infty}\frac{x^k\cos(k\pi/3)}{k}.$ |
| 8. | $Y(x)=e^{2x},$ | $S(x)=\sum_{k=0}^{\infty}\frac{(2x)^k}{k!}.$ |
| 9. | $Y(x)=\frac{1+x^2}{2}\operatorname{arctg}(x)-x/2,$ | $S(x)=\sum_{k=1}^{\infty}(-1)^{k+1}\frac{x^{2k+1}}{4k^2-1}.$ |
| 10. | $Y(x)=\frac{e^x+e^{-x}}{2},$ | $S(x)=\sum_{k=0}^{\infty}\frac{x^{2k}}{(2k)!}.$ |
| 11. | $Y(x)=(x^2/4+x/2+1)e^{x/2},$ | $S(x)=\sum_{k=0}^{\infty}\frac{k^2+1}{k!}(x/2)^k.$ |
| 12. | $Y(x)=(1-\frac{x^2}{2})\cos(x)-\frac{x}{2}\sin(x),$ | $S(x)=\sum_{k=0}^{\infty}(-1)^k\frac{2k^2+1}{(2k)!}x^{2k}.$ |
| 13. | $Y(x)=2(\cos^2 x-1),$ | $S(x)=\sum_{k=1}^{\infty}(-1)^k\frac{(2x)^{2k}}{(2k)!}.$ |

| | | |
|---|---|---|
| 14. | $Y(x)=\frac{e^{x}-e^{-x}}{2},$ | $S(x)=\sum_{k=0}^{\infty}\frac{x^{2k+1}}{(2k+1)!}.$ |
| 15. | $Y(x)=-\ln\sqrt{1+x^{2}}+x\operatorname{arctg}(x),$ | $S(x)=\sum_{k=1}^{\infty}(-1)^{k+1}\frac{x^{2k}}{2k(2k-1)}.$ |
| 16. | $Y(x)=\operatorname{arctg}(x),$ | $S(x)=\sum_{k=0}^{\infty}(-1)^{k}\frac{x^{2k+1}}{2k+1}.$ |

# ЛАБОРАТОРНА РОБОТА № 4.
# Масиви і файли в мові програмування С

## 4.1. Мета роботи

Навчитися використовувати масиви та файли при розв'язанні задач.

## 4.2. Теоретичні відомості

### 4.2.1.Масиви

Масив – це послідовно розміщені у пам'яті елементи однакового типу. Кожен масив має ім'я. Доступ до окремих елементів масиву відбувається по імені масиву та індексу (порядковому номеру, зміщенню відносно першого) елемента. Основні властивості масивів:

- всі елементи масиву мають однаковий тип;
- всі елементи масиву розміщені у пам'яті послідовно один за одним;
- індекс першого елемента рівний нулю;
- ім'я масиву є вказівником-константою, що вказує на адресу в пам'яті першого елемента масиву.

Ознакою масиву при описі є наявність парних квадратних дужок "**[ ]**". Константа або константний вираз у квадратних дужках задає число елементів. Розмір не може бути динамічним, тому не можна задавати його з допомогою змінних.

Одновимірний масив

Загальний вигляд стрічки оголошення одномірного масиву наступний:

```
<тип> <ідентифікатор>[<кількість елементів>];
```

Наприклад:

```
int x[10];
```


*Рис. 4.1 Розміщення елементів маству в пам'яті*

Загальний розмір масиву в байтах рівний розміру базового типу помноженого на кількість елементів. Для наведеного прикладу розмірність масиву рівна **sizeof(int)*10**. Приклад:

```
#include <stdio.h>
int main()
{
    int x[10];
    for(int i = 0; i < 10; i++)
    {   x[i] = i;
        printf("%d ", x[i]);
    }
    return 0;
}
```

Результат:

```
0 1 2 3 4 5 6 7 8 9
```

Двовимірні масиви

С дає змогу використовувати багатовимірні масиви. Найпростіший варіант – двовимірний масив. Для оголошення двовимірного масиву типу **int twodim** розмірністю **10** на **20** потрібно описати його наступним чином:

```
int twodim[10][20];
```

На відміну від інших мов програмування розмірності масиву відокремлені одна від одної квадратними дужками. Останній запис можна розглядати як масив масивів, тобто масив з двадцяти елементів типу **(twodim[10])**, що в свою чергу є масивом з десяти елементів типу **(int)**. Звідси випливає, що елементи багатовимірних масивів в мові С у пам'яті зберігаються по рядках.

Для доступу до елемента з індексами **3**, **5** масиву **twodim** служить запис **twodim[3][5]**. Наступний приклад демонструє роботу з двовимірним масивом, а саме, присвоєння кожному елементу цього масиву суму індексів:

```
#include <stdio.h>
int main()
{   int num[2][4];
    for(int i = 0; i < 2; i++)
    {   for(int j = 0; j < 4; j++)
        {   num[i][j] = i + j;
            printf("%d ", num[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Результат:

```
0 1 2 3
1 2 3 4
```

Двовимірні масиви можна розглядати як матриці, тобто у вигляді колонок та рядків. У пам'яті всі елементи багатовимірного масиву зберігаються підряд один за одним. Схематично це можна зобразити так (рис. 4.2).

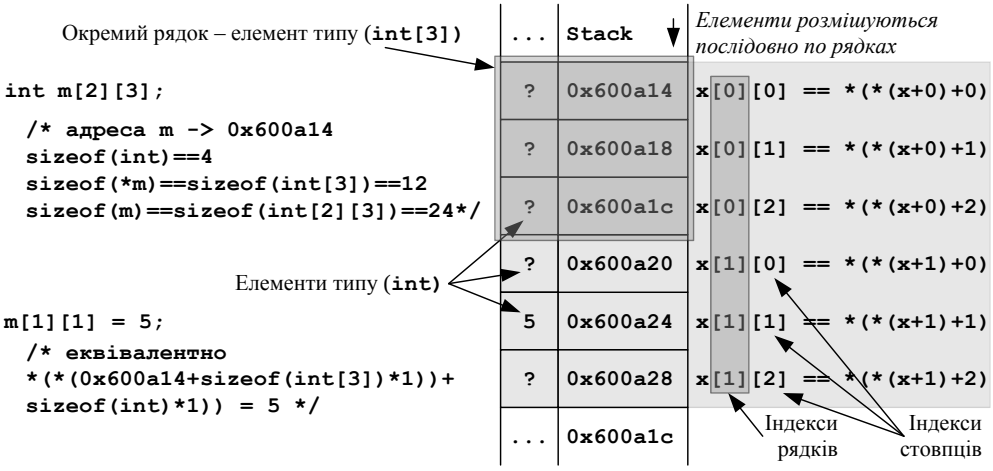



*Рис. 4.2 Розміщення елементів двовимірного маству в пам'яті*

Багатовимірні масиви

Загальний вигляд оголошення багатовимірного масиву наступний:

```
<тип> <ідентифікатор>[<розмір1>][<розмір2>]…[<розмірN>];
```

Наприклад :

```
int three[4][10][5];
int four[4][10][5][7];
```

Для масиву **four** необхідно **sizeof(int)*4*10*5*7** байт.

Ініціалізація масиву

С дає змогу ініціалізувати масиви при їх оголошенні. Загальна форма такої ініціалізації подібна до ініціалізації інших змінних:

```
<тип> <ім'я> [<розмір1>]…[<розмірN>] = {список значень};
```

Список значень – це розділений комами список констант. Наприклад:

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Це означає, що **i[0] = 1**, **i[1] = 2**, … , **i[9] = 10**.

Багатовимірні масиви ініціалізуються так само, як і одновимірні:

```
int s[4][2] = {
  1, 1,
  2, 4,
  3, 9,
  4, 16};
```

Для такої попередньої ініціалізації необхідно знати точну кількість елементів. Щоб не задавати фіксовану розмірність масиву, ініціалізацію можна провести наступним чином:

```
int i[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

а для багатомірного масиву:

```
int s[][2] = {1, 1,
              2, 4,
              3, 9};
```

Якщо кількість елементів для ініціалізації є завеликою, компілятор видасть повідомлення про помилку, якщо замалою – згідно стандарту, непроініціалізовані елементи заповняться нулями. Можлива ініціалізація окремих елементів масиву, наприклад:

```
int k[5] = {[2]=1};  //{0, 0, 1, 0, 0};
```

Стандартом не визначено, що міститимуть елементи непроініціалізованих масивів, якщо тільки вони не оголошені, як статичні – у такому випадку вони будуть містити нулі.

### 4.2.2. Файли, та робота з ними

Поняття файлу та файлової системи

*Файл* [10] – це набір даних в енергонезалежній пам'яті, до якого можна звертатися за іменем. Файли організовані у файлові системи. З погляду користувача та прикладних програм файл є мінімальним обсягом даних файлової системи, з яким можна працювати незалежно. Наприклад, користувач не може зберегти дані на зовнішньому носії, не звернувшись при цьому до файлу. Файли є найпоширенішим засобом зберігання інформації в енергонезалежній пам'яті. Така пам'ять надійніша, й інформація на ній може зберігатися так довго, як це необхідно.

*Файлова система* – це підсистема операційної системи, що підтримує організований набір файлів, здебільшого у конкретній ділянці дискового простору (логічну структуру); низькорівневі структури даних, використовувані для організації цього простору у вигляді набору файлів (фізичну структуру); програмний інтерфейс файлової системи (набір системних викликів, що реалізують операції над файлами). Файлова система надає прикладним програмам абстракцію файлу. Прикладні програми не мають інформації про те, як організовані дані файлу, як знаходять відповідність між ім'ям файлу і його даними, як пересилають дані із диска у пам'ять, тощо – усі ці операції забезпечує файлова система. Файлові системи можуть надавати інтерфейс доступу не тільки до диска, але й до інших пристроїв. Є навіть файлові системи, які не зберігають інформацію, а генерують її динамічно за запитом. Для прикладних програм усі такі системи мають однаковий вигляд.

Типи файлів

Раніше операційні системи підтримували файли різної спеціалізованої структури. Сьогодні є тенденція взагалі не контролювати на рівні операційної системи структуру файлів, відображаючи кожен простою послідовністю байтів. У цьому разі програми, які працюють із файлами, самі визначають їх формат.

Існують спеціальні файли, які операційна система інтерпретує особливим чином. Структуру таких файлів підтримується відповідно до тих задач, які з їхньою допомогою розв'язуються. Категорією таких файлів є виконувані файли. Хоч їх звичайно не розглядають разом зі спеціальними файлами, вони мають жорстко заданий формат, який розпізнає операційна система. Часто буває так, що система може працювати із виконуваними файлами різних форматів[1].

Іншим варіантом класифікації є поділ на файли із прямим і послідовним доступом. Файли із прямим доступом дають змогу вільно переходити до будь-якої позиції у файлі, використовуючи для цього поняття вказівника поточної позиції файлу (*seek pointer*), що може переміщатися у будь-якому напрямку за допомогою відповідних системних викликів. Файли із послідовним доступом можуть бути зчитані тільки послідовно, із початку в кінець. Сучасні операційні системи звичайно розглядають усі файли як файли із прямим доступом.

Імена файлів

Важливою складовою роботи із файлами є організація доступу до них за іменем. Різні системи висувають різні вимоги до імен файлів. Так, у деяких системах імена є чутливими до регістра ("**`myfile.txt`**" і "**`MYFILE.TXT`**" будуть різними іменами), а в інших – ні.

Операційна система може розрізняти окремі частини імені файлу. Кілька останніх символів імені (звичайно відокремлені від інших символів крапкою) у деяких системах називають *розширенням файлу*, яке може характеризувати його тип. В інших системах обов'язкове розширення не виділяють, при цьому деякі програми можуть, однак, розпізнавати потрібні їм файли за розширеннями (наприклад, компілятор C може розраховувати на те, що файли коду програм матимуть розширення "**`.c`**").

Важливою характеристикою файлової системи є максимальна довжина імені файлу. Багато операційних систем обмежували довжину імен файлів. Широко відоме було обмеження на 8 символів у імені та 3 – у розширенні, присутнє до появи Windows 95. Сьогодні стандартним значенням максимальної довжини імені файлу є 255 символів.

---

[1] Наприклад, Для лінійки Windows виконувані файли з розширенням "**`.exe`**" (*executable*) завжди починаються з двох байтів, в які записані символи "**`MZ`**"; У старих версіях Windows використовувалося позначення "**`NE`**"; Для OS/2 – "**`LE`**" або "**`LX`**"; для сучасних Windows – "**`PE`**". Наступні байти містять метаінформацію про програму (використання пам'яті, розмір стеку, початкові значення регістрів тощо), після яких слідує виконуваний код.

Каталоги

Для організації файлів було запропоновано поняття файлового каталогу (*file directory*) або просто каталогу. *Каталог* – це об'єкт (найчастіше реалізований як спеціальний файл), що містить інформацію про набір файлів. Про такі файли кажуть, що вони містяться в каталозі. Файли заносяться в каталоги користувачами на підставі їхніх власних критеріїв, деякі каталоги можуть містити дані, потрібні операційній системі, або її програмний код.

Каталог можна уявити собі як символьну таблицю, що реалізує відображення імен файлів у елементи каталогу (зазвичай в таких елементах зберігають низькорівневу інформацію про файли).

Базовою ідеєю організації даних за допомогою каталогів є те, що вони можуть містити інші каталоги. Вкладені каталоги називають підкаталогами (*subdirectories*). Таким чином формують дерево каталогів. Перший каталог, створений у файловій системі, встановленій у розділі (корінь дерева каталогів), називають кореневим каталогом (*root directory*).

Для файлу, розташованого всередині каталогу недостатньо його імені для однозначного визначення, де він перебуває – в іншому каталозі може бути файл із тим самим ім'ям. Для визначення місцезнаходження файлу потрібно додавати до його імені список каталогів, де він перебуває. Такий список називають шляхом (*path*). Каталоги у шляху перераховують зліва направо – від меншої глибини вкладеності до більшої. Роздільник каталогів у шляху відрізняється для різних систем: в UNIX прийнято використовувати прямий слеш "**/**", а у Windows-системах – зворотний "****"[1].

Є два шляхи до файлу: абсолютний і відносний. *Абсолютний* (або *повний*) повністю й однозначно визначає місце розташування файлу. Такий шлях обов'язково має містити кореневий каталог. Ось приклад абсолютного шляху для UNIX-систем: "**/usr/local/bin/myfile**". Якщо програма використовує тільки абсолютні шляхи, їй зазвичай бракує гнучкості. Наприклад, у разі перенесення в інший каталог потрібно буде вручну відредагувати всі шляхи, замінивши їх новими.

*Відносний* – шлях, відлічуваний від деякого місця в ієрархії каталогів. Щоб його організувати, потрібно визначитися із точкою відліку, для чого використовують поняття поточного каталогу. Такий каталог задають для кожного процесу (програми, що виконується). Відносний шлях може відлічуватися від поточного каталогу і зазвичай кореневий каталог не включає. Прикладом відносного шляху до файлу "**/usr/local/bin/myfile**" (за

[1] Нагадаємо, що оскільки компілятор мови С при зустрічі символу "****" інтерпретує його як початок керуючої послідовності, для правильного вказання шляхів у Windows системах необхідно подвоювати символ, наприклад **"C:\\myprogram\\myfile.exe"**.

умови, що поточним каталогом є "**/usr/local**") буде "**bin/myfile**", а в ситуації, коли поточним є каталог файлу ("**/usr/local/bin**"), відносним шляхом буде просто ім'я файлу.

Для спрощення побудови відносного шляху кожний каталог містить два спеціальні елементи:

- "**.**", що посилається на поточний каталог;
- "**..**", що посилається на каталог рівнем вище.

З урахуванням цих елементів можуть бути задані такі відносні шляхи, як "**../../bin/myfile**" (за умови, що поточний каталог – "**/usr/local/lib/mylib**") або "**./myfile**" (вказує на елемент у поточному каталозі). Програми, що обмежуються тільки відносними шляхами під час доступу до файлів (особливо, якщо вони не виходять за межі каталогу цих програм), можуть бути без змін перенесені в інші каталоги тієї самої структури.

Текстовий і бінарний режим

У мові С файл розглядається як безперервна послідовність байтів, кожен з яких може бути прочитаний індивідуально. Це відповідає файловій структурі системи UNIX, звідки С бере свій початок. Оскільки інші системи можуть не відповідати в точності цій моделі, в С пропонуються два способи представлення файлів: *текстовий режим* і *бінарний режим*.

*Текстові* файли можуть бути переглянуті й відредаговані із клавіатури будь-яким текстовим редактором і мають дуже просту структуру: послідовність ASCII-символів. Ця послідовність символів може бути розбита на рядки, кожна з яких закінчується двома службовими символами з кодами: **13**, **10** (**0xD**, **0xA**), що відповідають спеціальним символам повернення каретки на початок рядка **'\r'** та переходу на новий рядок **'\n'** відповідно. У кінці файлу обов'язково присутній службовий символ кінця файлу **EOF**, що для операційних систем Windows відповідає комбінації клавіш "**Ctrl+Z**", а для UNIX та Linux систем – "**Ctrl+D**". Приклади відомих текстових файлів: "***.bat**", "***.c**", "***.txt**".

*Бінарні* файли – це файли, які не мають структури текстових файлів. Кожна програма для своїх бінарних файлів визначає власну структуру. З будь-яким файлом можна працювати як у бінарному, так і в текстовому режимах. Якщо текстовий файл читати у бінарному режимі всі службові символи втрачають свій сенс, наприклад символи переходу на новий рядок та перевід на початок каретки будуть розглядатися просто як два байти зі значеннями **0xD**, **0xA**.

Рівні вводу-виводу

На додаток до вибору представлення файлу в більшості випадків можна вибрати один з двох рівнів вводу-виводу (тобто з двох рівнів управління доступом до файлів). *Низькорівневий ввід-вивід* передбачає використання

основних служб вводу-виводу, що надаються операційною системою. Стандартний *високорівневий ввід-вивід* передбачає застосування стандартного пакету бібліотечних функцій С та визначень бібліотеки **<stdio.h>**. Стандарт С підтримує тільки стандартний пакет вводу-виводу, тому що немає ніякої можливості гарантувати, що всі операційні системи можуть бути представлені однаковою низькорівневою моделлю вводу-виводу.

Потоковий ввід-вивід

У порівнянні з низькорівневим вводом-виводом стандартний пакет вводу-виводу, крім переносимості, володіє ще двома перевагами. По-перше, в ньому доступно багато спеціалізованих функцій, які спрощують вирішення різноманітних завдань, пов'язаних з вводом-виводом. Наприклад, функція **printf()** перетворює різні форми даних в символьні рядки. По-друге, ввід і вивід є буферизованим. Це означає, що інформація передається великими порціями (зазвичай по 512 і більше байт), а не по одному байту за раз. Наприклад, коли програма читає файл, порція даних зчитується в буфер – проміжну область пам'яті. Така буферизація істотно збільшує швидкість передачі даних. Потім програма може використовувати окремі байти в цьому буфері. Буферизація відбувається "за кулісами", тому створюється ілюзія посимвольного доступу.

Буферизація реалізована на концепції потокового вводу-виводу. Програми спроектовані за допомогою мови С можуть використовуватися для роботи з великою кількістю пристроїв: моніторами, дисководами, DVD-ROM, принтерами, тощо. Незважаючи на те, що кожний з цих пристроїв має свої особливості, програма на С розглядає їх як деякий логічний пристрій або файл[1]. Таку абстракцію називають *потоком*.

Всі потоки однакові за поведінкою, але аналогічно до файлів, розрізняють два типи потоків – бінарні і текстові. Текстовий потік – послідовність ASCII-символів, бінарний – послідовність байтів.

Потік асоціюється з певним файлом при його відкритті. Коли файл є відкритим, між ним і програмою можливий обмін інформацією. Далі наведено декілька основних функцій для роботи з файлами:

| | | |
|---|---|---|
| **fopen()** | – | відкриває потік; |
| **fclose()** | – | закриває потік |
| **fprintf()** | – | запис у файл; |

[1] Концепція прийшла з операційної системи UNIX і в оригіналі звучить як "*Everything is a file*" – описує одну з визначальних рис UNIX і похідних систем, де широкий діапазон вхідних вихідних ресурсів, таких як документи, каталоги, жорсткі диски, модеми, клавіатури, принтери і навіть деякі міжпроцесорні та мережеві комунікації розглядаються як прості потоки байтів. Перевага такого підходу полягає в тому, що той ж набір інструментів, може бути використаний в широкому спектрі ресурсів.

| | | |
|---|---|---|
| **fscanf()** | – | читання з файлу; |
| **feof()** | – | повертає істинне значення, якщо досягнуто кінець файлу; |
| **remove()** | – | знищує файл. |

Відкриття файлу

Для початку роботи з файлом його необхідно "*відкрити*". Функція відкриття файлу **fopen()** має два параметри, обидва є стрічковими літералами:

```
FILE *fopen(char *filename, char *mode);
```

Перший – задає шлях[1] та ім'я файлу, що відкривається, а другий – тип доступу до файлу, що може приймати наступні значення:

| | |
|---|---|
| **"r"** | Відкрити файл для читання |
| **"w"** | Відкрити файл для запису. Якщо файл існує, то його вміст знищується |
| **"a"** | Відкрити файл для запису в кінець файлу. Якщо файл не існує, то він створюється |
| **"r+"** | Відкрити файл для читання й запису. Файл повинен існувати |
| **"w+"** | Відкрити файл для читання й запису. Якщо файл існує, то його вміст знищується |
| **"a+"** | Відкрити файл для читання й запису в кінець файлу. Якщо файл не існує, то він створюється |

До комбінацій перерахованих літералів можуть бути *додані* також **"t"** або **"b"**:

| | |
|---|---|
| **"t"** | Відкрити файл у текстовому режимі |
| **"b"** | Відкрити файл у бінарному режимі |

Можливі *режими доступу:* **"w+b", "wb+", "rw+", "w+t", "rt+"** і ін. Якщо режим *не зазначений,* то файл відкривається в *текстовому* режимі.

Функція **fopen()** не тільки відкриває файл, а й налаштовує буфер (або два буфера для режимів читання-запису) і встановлює структуру даних **FILE**, що містить відомості про файл і про буфер. Крім того, **fopen()** повертає вказівник на цю структуру, так що інші функції знають, де її шукати[2].

---

[1] Якщо програма запускається на виконання з консолі, то вона очікує, що файл з вказаним ім'ям знаходиться в тому ж каталозі, що і програма. Якщо програма запускається з IDE-середовища, то каталог, в якому проводиться пошук файлу, залежить від реалізації. Наприклад, за замовчуванням Microsoft Visual Studio переглядає каталог, що містить вихідний код, а XCode шукає файл в каталозі, де розташований виконуваний файл. IDE Code::Blocks переглядає файли у каталозі проекту. Також у всіх випадках пошук здійснюється у каталогах, що прописані у системній змінній "**PATH**".

[2] Керування відкритими файлами здійснюється операційною системою за допомогою таблиці дескрипторів (**handle**). *Файловий дескриптор* це абстрактний вказівник для доступу до відкритого файлу або іншого пристрою вводу-виводу. Файловий дескриптор є невід'ємним

Наприклад:

```
FILE *fp;                  //file pointer declaration
fp = fopen("test", "w"); //open file for writing
```

Тут **"w"** означає *writing* – запис. Для читання з файлу потрібно використовувати атрибут **"r"**– *reading.*

Стандартні файли

Програми на мові С автоматично відкривають три потоки, які називаються *стандартним вводом*, *стандартним виводом* і *стандартним виводом помилок*. У бібліотеці **<stdio.h>** для них зарезервовані файлові вказівники "**stdin**", "**stdout**" і "**stderr**", відповідно. За замовчуванням стандартний ввід є звичайним пристроєм вводу у системі, як правило – клавіатура. Стандартний вивід і стандартний вивід помилок за замовчуванням є звичайним пристроєм виведення системи, тобто – екран монітора.

Природно, стандартний ввід забезпечує введення даних в програму. Це потік, який читається за допомогою функцій **getchar()** і **scanf()**. Стандартний вивід – місце, куди направляється звичайний вивід програми. Він використовується такими функціями як **putchar()** і **printf()**. Призначення потоку стандартного виводу помилок полягає в тому, щоб визначити логічно відокремлене місце для виводу повідомлень про помилки. Всі стандартні потоки можна перенаправити.

Закриття файлу

Після роботи з файлом він повинен бути *закритий* функцією **fclose()**. Для цього необхідно в зазначену функцію передати вказівник на **FILE**, що був отриманий при відкритті функцією **fopen()**. У процесі завершення програми незакриті файли автоматично закриваються системою, всі буферизовані операції запису примусово виконуються, так щоб очистити буфери і записати необхідну інформацію безпосередньо у файли. Файл, відкритий у попередньому прикладі, можна закрити так:

```
fclose(fp);
```

Якщо файл не закрити, то не можна гарантувати, наприклад, завершення буферизованого запису інформації в нього. Обійти цю незручність можна примусовим записом з допомогою виклику функції **fflush()**.

Читання/запис у файл

Для роботи з текстовими файлами в бібліотеці мови С є низка зручних функцій, розглянемо найпоширеніші: **fprintf()**, **fscanf()**. Формат параметрів-цих функцій дуже схожий на формат знайомих функцій **printf()**, **scanf()**. Схожі не тільки параметри, але й дії. Відмінність

---

цілим числом, як правило типу **int**. Дескриптор міститься у структурі типу **FILE**.

складається лише в тому, що **printf()** і **scanf()** працюють за замовчуванням з **stdin** та **stdout**, а **fprintf()** і **fscanf()** – з явно вказаними файлами (у тому числі й з **stdin** та **stdout**), тому в них доданий параметр, що є вказівником на структуру **FILE**.

Функції **fscanf()** та **fprintf()**

Зв'язавши ці функції з певним потоком, можна використовувати їх, як звичайні функції **printf()** та **scanf()**. Наприклад:

```
#include <stdio.h>
int main(){
    //open file for writing,
    //create if doesn't exist
    FILE *myFile = fopen("test.dat", "w");
    int i, x[10];
    for(i = 0; i < 10; i++)
        //write into the file
        fprintf(myFile, "%d ", i);
    //close file - flush all
    //buffered data into it
    fclose(myFile);
    //open file for reading
    myFile = fopen("test.dat", "r");
    for(i = 0; i < 10; i++)
        //read data from the file
        fscanf(myFile, "%d ", &x[i]);
    fclose(myFile);
    //remove the file from filesystem
    remove("test.dat");
    return 0;
}
```

Якщо вивчити вміст файлу, створеного цією програмою, то видно, що дані в ньому розташовуються так само, як розташовувалися б на екрані, якби використовувалася функція **printf()**.

Функції **fread()** та **fwrite()**

Функції для роботи з *текстовими* файлами зручно використовувати при створенні текстових файлів, веденні файлів-протоколів (log-файлів) і т. п. Але часто при вирішенні задач, наприклад при створенні *баз даних* доцільно використовувати функції для роботи з *бінарними* файлами: **fwrite()** і **fread()**. Ці функції без яких-небудь змін *копіюють* блок даних з оперативної пам'яті у файл і, відповідно, з файлу – в пам'ять. Такий спосіб обміну даними вимагає менше часу. Прототипи функцій:

```
int fread(
  void *ptr,
  size_t size,
  size_t count,
  FILE *stream);
int fwrite(
  const void *ptr,
  size_t size,
  size_t count,
  FILE * stream);
```

де[1]:

***ptr** – вказівник на буфер;
**size** – розмір блоку;
**count** – кількість блоків;
***stream** – вказівник на структуру **FILE** відкритого потоку.

Першим параметром передається вказівник на буфер, у який будуть поміщені дані з файлу функцією **fread()** або з якого дані будуть прочитані у файл функцією **fwrite()**. Наступні два параметри задають розмір блоку в байтах і відповідно, кількість блоків, що читаються/записуються. Останній параметр – вказівник на структуру **FILE**. Тобто, функція **fread()** читає **count** елементів даних з потоку **stream**, де кожен елемент має розмір **size** байт і зберігає їх в буфері **ptr** і повертає кількість прочитаних байт (**size*count**), або (**-1**) у випадку помилки. Функція **fwrite()** записує **count** елементів даних з буферу **ptr** в потік **stream**, де кожен елемент має розмір **size** байт і повертає кількість записаних байт (**size*count**), або (**-1**) у випадку помилки.

Стандартні функції введення-виведення орієнтовані на текст, працюючи з символами і рядками. А що, якщо в файлі потрібно зберегти числові дані? Дійсно, можна скористатися функцією **fprintf()** і форматом **%f**, щоб зберегти значення з плаваючою комою, але тоді воно збережеться як послідовність символів. Наприклад, код:

```
float num = 1./3.;
fprintf(fp, "%f", num);
```

зберігає **num** у вигляді послідовності з восьми символів: **0.333333**. Застосування специфікатора **%.2f** дає змогу зберегти його як послідовність з чотирьох символів: **0.33**. Використання специфікатора **%.12f** дає можливість

[1] **size_t** (зазвичай заданий через **#define** як **unsigned long**) – стандартний цілочисельний тип, що для конкретної платформи здатен вмістити максимальний розмір об'єкту будь-якого типу, в тому числі і масивів.

зберегти його у вигляді 14 символів: **0.333333333333**. Зміна специфікаторів приводимт до зміни розміру простору, необхідного для значення. Після того як значення **num** було збережено як **0.33**, немає ніякої можливості повернуться до повної точності значення при його читанні з файлу. У загальному випадку функція **fprintf()** претворює числові значення в символьні дані, можливо змінюючи значення.

Найбільш точний і однозначний спосіб збереження числа передбачає використання того ж самого набору бітів, що і комп'ютер. Таким чином, значення **float** має бути збережено в області з розміром як у типу **float.** Коли дані зберігаються в файлі в представленні, яке застосовується в програмі, говоримо, що дані збережені в бінарній формі. Ніякі перетворення з числових форм в символьні не відбуваються. Наприклад:

```
#include <stdio.h>
int main()
{
    float num = 1./3.;
    float res;
    FILE *fp = fopen("f.txt", "w");
    fprintf(fp, "%f", num);
    fclose(fp);
    fp = fopen("f.txt", "r");
    fscanf(fp, "%f", &res);
    fclose(fp);
    printf("Text mode\n"
           "num = %f, res = %f, "
           "(num = res) == %d\n",
           num, res, num == res);
    fp = fopen("f.txt", "wb");
    fwrite(&num, sizeof(float), 1, fp);
    fclose(fp);
    fp = fopen("f.txt", "rb");
    fread(&res, sizeof(float), 1, fp);
    printf("Binary mode\n"
           "num = %f, res = %f, "
           "(num = res) == %d\n",
           num, res, num == res);
    fclose(fp);
    return 0;
}
```

Результати:

```
Text mode
num = 0.333333, res = 0.333333, (num = res) == 0
Binary mode
num = 0.333333, res = 0.333333, (num = res) == 1
```

Насправді, всі дані зберігаються в двійковій формі. Навіть символи зберігаються з використанням двійкового представлення їх кодів відповідно до спеціальної таблиці смволів (ASCII-таблиці). Однак, якщо всі дані у файлі інтерпретуються як коди символів, говоримо, що файл містить текстові дані. Якщо деякі або всі дані інтерпретуються як числові дані в двійковій формі, говоримо, що файл містить бінарні дані. Наприклад, файли, що генеруються текстовими процесорами типу MS Word, як правило, є двійковими, оскільки вони містять багато нетекстової інформації, що описує наприклад шрифти, форматування та інші деталі.

Важливо зрозуміти, що немає однозначного методу, який можна було б використовувати для того, щоб розрізнити текстовий файл від бінарного, тому будь-який бінарний файл може бути відкритий для роботи з ним як з текстовим; а текстовий може бути відкритий як бінарний. Але такий варіант роботи з файлом звичайно приводить до помилок. Тому рекомендується працювати з конкретним файлом у тому режимі, у якому він був створений.

Використання функції **feof()**

Функція **feof()** повертає істинне значення у випадку досягнення кінця файлу (знаходження **EOF**), і нуль – в протилежному випадку. Наприклад у попередньому прикладі операція зчитування з файлу може виглядати наступним чином:

```
i = 0;
while(!feof(myFile))
    scanf(myFile, "%d ", &x[i++]);
```

Наприклад, з файлу прочитати матрицю розмірності 12×12. Порахувати суму елементів над головною діагоналлю, які більші 7:

```
#include <stdio.h>
int main()
{
    int a[12][12], s = 0;
    FILE *fp = fopen("f.txt", "r");
    printf("Begin:\n");
    printf("s = %d \n", s);
    for(int i = 0; i < 12; i++)
        for(int j = 0; j < 12; j++)
            fscanf(fp, "%d", &a[i][j]);
    printf("The matrix:\n");
```

```
        for(int i = 0; i < 12; i++)
        {
            for(int j = 0; j < 12; j++)
                printf("%d ", a[i][j]);
            printf("\n");
        }
        for(int i = 0; i < 12; i++)
            for(int j = 0; j < 12; j++)
                if((j - i >= 0) && (a[i][j] > 7))
                    s += a[i][j];
        fclose(fp);
        printf("The sum: \n");
        printf("s = %d\n", s);
        return 0;
    }
```

Результати:

```
    Begin:
    s = 0
    The matrix:
    1 7 4 0 9 4 8 8 2 4 5 5
    1 7 1 1 5 2 7 6 1 4 2 3
    2 2 1 6 8 5 7 6 1 8 9 2
    7 9 5 4 3 1 2 3 3 4 1 1
    3 8 7 4 2 7 7 9 3 1 9 8
    6 5 0 2 8 6 0 2 4 8 6 5
    0 9 0 0 6 1 3 8 9 3 4 4
    6 0 6 6 1 8 4 9 6 3 7 8
    8 2 9 1 3 5 9 8 4 0 7 6
    3 6 1 5 4 2 0 9 7 3 7 2
    6 0 1 6 5 7 5 4 1 2 0 0
    1 4 6 0 7 1 7 7 7 7 3 3
    The sum:
    s = 118
```

Позиціонування у файлі

Кожен відкритий файл має так званий *вказівник* на поточну позицію у файлі (подібно до вказівника в пам'яті). Всі операції над файлами (читання й запис) починають виконуватися із цієї позиції. При кожному виконанні функції читання або запису вказівник зміщається на кількість прочитаних або записаних байт, тобто встановлюється відразу за прочитаним або записаним блоком даних

у файлі. У цьому випадку здійснюється, так званий, *послідовний доступ* до даних, що дуже зручно, коли необхідно послідовно працювати з даними у файлі. Це демонструється у всіх вищенаведених прикладах читання й запису у файл. Але іноді необхідно зчитувати або записувати дані в довільному порядку, що можливо шляхом установки вказівника на деяку задану позицію у файлі функцією **fseek()**:

```
int fseek(FILE *stream, long offset, int whence);
```

Параметр **offset** задає кількість байт, на яку необхідно змістити вказівник відповідно параметру **whence**. Значення, які може приймати параметр **whence**:

| | |
|---|---|
| **SEEK_SET** | Зсув виконується від початку файлу |
| **SEEK_CUR** | Зсув виконується від поточної позиції вказівника |
| **SEEK_END** | Зсув виконується від кінця файлу |

Величина зсуву може бути як додатною, так і від'ємною, але не можна зміститися за межі початку файлу. Такий доступ до даних у файлі називають *довільним*.

Іноді необхідно визначити позицію вказівника. Для цього можна скористатися функцією **ftell()**, що повертає значення вказівника на поточну позицію файлу. У випадку помилки повертає число (**-1**).

Деякі функції бібліотеки мови С для роботи з файлами

```
char * fgets(char *string, int n, FILE *stream);
```

зчитує рядок з файлу **stream** доти, поки не будуть зчитані (**n-1**) символів, або символ переходу рядка, або досягнутий кінець файлу. Повертає вказівник на **string**, а у випадку помилки або при досягненні кінця файлу – **NULL**.

```
int fputs(char *string, FILE *stream);
```

копіює у файл символьний рядок з пам'яті, на яку вказує **string**. Повертає код останнього записаного у файл символу, при помилці – **EOF**.

```
int fsetpos(FILE *stream, const long *pos);
```

установлює значення вказівника читання/запису (вказівник на поточну позицію) файлу в позицію, задану значенням по вказівнику **pos**. Повертає **0** при коректному виконанні й будь-яке ненульове значення при помилці.

```
int fgetpos(FILE *stream, long *pos);
```

поміщає в змінну, на яку вказує **pos**, значення вказівника на поточну позицію у файлі. Повертає **0** при коректному виконанні й будь-яке ненульове значення при помилці.

```
int fflush(FILE *fp);
```

подібно до операції закриття файлу призводить до того, що будь-які незаписані дані в буфері виведення відправляються у вихідний файл, ідентифікований за

допомогою **fp**, але файл при цьому залишається відкритим. Цей процес називається скиданням буфера. Якщо **fp** – нульовий вказівник, то скидаються всі буфери виводу. Результат використання функції **fflush()** на вхідному потоці не визначено.

Низькорівневий ввід/вивід

У мові С є можливість обробки файлів, використовуючи дескриптори. *Дескриптор* (**handle**) – це ціле число, що система ставить у відповідність відкритому файлу й надалі використовує в операціях роботи з файлом. Нижче наводиться опис деяких функцій, що працюють із дескрипторами. Для застосування цих функцій необхідно підключити до програмного модуля бібліотеку **<io.h>**. Увага! Даний функціонал не є частиною стандарту, тому його підтримка залежить від конкретного компілятора;

```
int open(char *filename, int access [, unsigned mode]);
```

функція виконує відкриття файлу. Повертає дескриптор файлу або (**-1**) при невдалому завершенні операції. Параметр **access** задає режими доступу до файлу. При роботі з дескрипторами необхідно підключати заголовний файл **<fcntl.h>,** у якому описані нижче вказані константи.

| | |
|---|---|
| **O_RDONLY** | Відкрити тільки для читання |
| **O_WRONLY** | Відкрити тільки для запису |
| **O_RDWR** | Відкрити для читання/запису |
| **O_CREAT** | Створити й відкрити файл. Якщо файл існує, то він просто відкривається |
| **O_EXCL** | Використовується тільки з **O_CREAT**. Якщо файл існує, то операція перерветься з помилкою |
| **O_TRUNC** | При відкритті урізати довжину існуючого файлу до нуля |
| **O_BINARY** | Відкрити файл у бінарному режимі |
| **O_TEXT** | Відкрити файл у текстовому режимі |

```
int close(int handle);
```

закриває файл, відкритий функцією **open()**. Повертає **0**, якщо закриття пройшло успішно, інакше – (**-1**).

```
long keekf(int handle, long offset, int fromwhere);
```

функція встановлює вказівник запису/читання файлу на позицію **offset** у напрямку, зазначеному **fromwhere** (див. **fseek**).

```
long tell(int handle);
```

повертає в байтах (від початку файлу) поточну позицію вказівника запису/читання. У випадку помилки – (**-1**).

```
long filelength(int handle);
```

повертає розмір файлу в байтах.

```
int chsize(int handle, long newsize);
```

функція виконує зміну розміру файлу. **EOF** встановлюється після байту з номером **newsize**. Повертає **0**, якщо розмір файлу змінений, інакше – (**-1**).

```
int read(int handle, void *buf, unsigned len);
```

читає **len** байт з файлу в область пам'яті, на яку вказує **buf**. Повертає кількість прочитаних байтів.

```
int write(int handle, void *buf, unsigned len);
```

записує **len** байт у файл із області пам'яті, на яку вказує **buf**. Повертає кількість записаних байтів.

У ряді задач необхідно перейти від вказівника на файл до дескриптора файлу. У таких випадках використовується функція:

```
int fileno(FILE *stream);
```

функція повертає дескриптор файлу, пов'язаний з вказівником на файл **stream**. Наприклад, для визначення довжини файлу:

```
#include <stdio.h>
int main()
{
    FILE *fp = fopen("f.txt", "r");
    int fd = fileno(fp);    //file handle
    printf("File length is %ld bytes\n",
        filelength(fd));
    fclose(fp);
    return 0;
}
```

Результати:

```
File length is 312 bytes
```

### 4.3. Контрольні запитання

1. Що таке масив?
2. Як обчислити розмір масиву?
3. В якій послідовності розміщуються в пам’яті елементи двовимірного масиву?
4. Що таке файл?
5. Що таке шлях до файлу?
6. Що таке текстовий і бінарний режим роботи з файлами?
7. Які Ви знаєте рівні роботи з файлами?
8. Що таке потоковий ввід-вивід?
9. Які існують стандартні потоки?
10. Які параметри функції **`fopen()`**?
11. Що таке дескриптор?
12. Які функції використовуються для роботи з файлами в текстовому режимі?
13. Які функції використовуються для роботи з файлами в бінарному режимі?

### 4.4. Лабораторне завдання

1. Навчитися використовувати масиви та файли при розв’язанні задач.
2. Одержати індивідуальне завдання.
3. Побудувати блок-схеми алгоритмів відповідно до завдання.
4. Скласти програми на алгоритмічній мові С згідно завдання.
5. Відлагодити програми, виконати обчислення, проаналізувати отримані результати.

### 4.5. Зміст звіту

1. Титульний аркуш.
2. Мета роботи.
3. Індивідуальне завдання.
4. Блок-схема алгоритмів у відповідності до завдання.
5. Тексти програм у відповідності до завдання.
6. Результати обчислень.
7. Аналіз результатів, висновки.

### 4.6. Індивідуальні завдання

Завдання 1

Дано натуральне число $N$ (задати довільно, як константу препроцесора) і одновимірний масив $A_0, A_1, \ldots, A_{N-1}$ цілих чисел (згенерувати додатні та від'ємні елементи випадковим чином, за допомогою функції бібліотеки **<stdlib.h> rand()**). Виконати наступні дії:

1. Визначити найбільше з непарних і кількість парних чисел.
2. Одержати масив, що відрізняється від оригінального тим, що всі непарні за порядком елементи подвоєні, а парні отримані додаванням власного значення з початковим значенням наступного елемента.
3. Визначити три максимальних і два мінімальних значення цього масиву.
4. Визначити найменше додатне.
5. Визначити число пар двох однакових додатних чисел, наприклад, чотири числа утворюють дві пари.
6. Визначити число пар двох однакових чисел різного знаку, наприклад, чотири числа утворюють дві пари.
7. Для кожного елемента визначити число його входжень у даний масив.
8. Одержати всі елементи, що входять у даний масив тільки по одному разі.
9. Одержати всі елементи, що входять у даний масив більше одного разу.
10. Визначити найбільше й найменше значення.
11. Замінити елементи, розташовані в парних позиціях першої половини масиву, подвоєними значеннями елементів, розташованих у непарних позиціях другої половини масиву.
12. Знайти найменший елемент у найбільш довгій безперервній послідовності додатних значень.
13. Виконати сортування елементів по зростанню.
14. Замінити всі нульові елементи масиву найменшим по модулю, але відмінним від нуля елементом.
15. Поміняти місцями елементи з мінімальним і максимальним значеннями.
16. Замінити від'ємні елементи масиву на середнє арифметичне значення чисел цього масиву.

Завдання 2

У файл **F1.txt** попередньо записати матрицю цілих чисел $A(N,N)$ (згенерувати випадковим чином, $N$ задати довільно, як константу препроцесора). Прочитати матрицю з файлу, виконати описані нижче дії, їх результати записати в файл **F2.txt**.

1. Переставити стовпці матриці у зворотному порядку.
2. Поміняти місцями елементи головної і бічної діагоналей.

3. Визначити найменші у своєму рядку елементи матриці та їх індекси.
4. Знайти номер рядка, всі елементи якого є парними числами (якщо не відається згенерувати випадково відповідну матрицю, то задати її явно).
5. Транспонувати матрицю.
6. Відсортувати кожен рядок по зростанню.
7. Знайти суму елементів розміщених над головною діагоналлю.
8. Обчислити суму від'ємних елементів кожного рядка.
9. Визначити чи є ця матриця симетричною відносно головної діагоналі.
10. Обчислити скалярний добуток рядка в якому міститься найбільший елемент матриці на стовпець з найменшим елементом.
11. Пронормувати кожен рядок цієї матриці поділивши на найбільший по модулю елемент.
12. Знайти суму додатніх елементів кожного рядка.
13. Обчислити середнє арифметичне додатніх елементів кожного стовпця.
14. Знайти суму додатніх елементів, які розміщені під головною діагоналлю і суму від'ємних елементів, які розміщені над головною діагоналлю.
15. Знайти максимальний елемент матриці і обнулити рядок та стовпець, в якому він знаходиться.
16. Обчислити середнє арифметичне всіх додатніх елементів, що знаходяться під головною діагоналлю.

# ЛАБОРАТОРНА РОБОТА № 5.
# Масиви символів (рядки) в мові програмування С

## 5.1. Мета роботи

Навчитися використовувати символьні масиви для розв'язання задач роботи зі стрічками.

## 5.2. Теоретичні відомості

### 5.2.1.Рядки символів і дії з ними

На відміну від інших мов програмування у С не визначено спеціального типу для опрацювання рядків. Масив символів (чи рядок або стрічка) розглядається як масив елементів типу **char**, який закінчується символом **'\0'** (нуль-символ) що є ознакою кінця рядка. Такі рядки називають ASCII-рядками. Сталі типу рядок записують у лапках, наприклад, **"Lviv Polytechnic"**, **"students"**, **" "** – рядок, що містить один символ-пробіл. У таких сталих рядках нуль-символ дописується автоматично.

Масиви символів оголошують так:

```
char <назва рядка>[довжина рядка];
```

Підчас оголошення символьного масиву необхідно до фактичної довжини рядка додати одиницю для нульового символу. Якщо масив символів оголошують й ініціалізують одночасно, то довжину можна не зазначати, компілятор визначить її сам. Оскільки рядки є масивами символів, то назва рядка є вказівником-константою на його перший елемент (на перший символ).

Розглянемо оголошення та ініціалізацію рядків:

```
const char text1[] = "We are learning C language";
char text2[] = "We" " are learn"
   "ing C language ";
char word[] = "University";
char lexem1[11], lexem2[40];
```

Тут оголошено константу **text1**, яка має значення **"We are learning C language"**, символьні масиви: **word** (без зазначення розміру), **lexem1** (може містити до 10 символів) та **lexem2** (до 39 символів).

Символьний масив **word** ще можна оголосити так:

```
char word[11] = "University";
```

або так:

```
char word[ ] = {
   'U','n','i','v','e','r','s','i','t','y','\0'};
```

Тут потрібно *вручну* записати нуль-символ, інакше змінна **word** трактуватиметься не як рядок, а як звичайний масив, і при спробі його вивести,

на екран, зазвичай буде виведено все "сміття", що знаходиться в пам’яті після адрес відведених під рядок, аж поки не зустрінеться перший нуль-символ. Наприклад:

```
double pi = 3.14159;
char word[10] = {
   'U','n','i','v','e','r','s','i','t','y'};
printf("%s", word);
```

Виведе щось подібне до:

```
UniversitynЖ←Ё·!        @
```

Це відбулося тому, що компілятор розмістив в наступних комірках число **pi**, а функція **printf()** спробувала вивести його на екран побайтно у вигляді символів.

Рядки можна опрацьовувати посимвольно за допомогою назви масиву, наприклад, так:

```
for (int n = 0; n < 11; n++)
     *(lexem1+n) = *(word+n);
printf("%s", lexem1);
```

Змінній **lexem1** надається значення **"University"** і ця фраза виводиться на екран. Інакше це можна зробити так:

```
for (int n = 0; n < 11; n++) lexem1[n] = word[n];
printf("%s", lexem1);
```

Ввести весь масив символів можна за допомогою команди (**'\0'** додається автоматично):

```
scanf("%s", <назва масиву>);
```

Якщо рядок даних містить символ пропуску (**' '**, **'\t'**, **'\n'**, **'\v'**, **'\f'**, **'\r'**), то **scanf()** зчитає дані лише до першого пропуску. Щоб ввести весь рядок до символу вводу, можна або задати спеціальне форматування з явним вказанням символів, що необхідно ігнорувати, або застосувати функції бібліотеки **<stdio.h>**:

```
char * gets(char * str);
int puts(const char *str);
```

При вдалому виконанні функція **gets()** повертає рядок, при невдалому нуль. Також нуль буде повернуто, якщо у рядку зустрінеться **EOF.**

Обидві функції не перевіряють розмір буферу, куди буде записаний рядок, тому відповідальність за можливе переповнення покладається на програміста. Більше того, стандарт C11 вважає функції типу **gets()** та **puts()** застарілими та строго не рекомендує їх використовувати. Для уникнення таких випадків можна використовувати функції для роботи з файлами **fgets()** та **fputs()**:

```
char * fgets( char * str, int num, FILE * stream);
```

```
int fputs(const char * str, FILE * stream );
```

**fputs()** повертає істине значення при вдалому виклику і **EOF** при невдалому.

Слід зауважити, що ці функції орієнтовані на роботу з файлами, тому мають ряд особливостей:

- **fgets()** приймає другим аргументом максимальну кількість символів для читання. Читання відбувається до першого символу переходу на новий рядок або до максимальної кількості мінус один символ;
- якщо **fgets()** зустрічає символ нового рядка, вона дописує його у відведений буфер на відміну від **gets()**, яка його просто відкидає, але **fputs()** не додає символ переходу на новий рядок у вихідні дані, на відміну від **puts()**.

Третім аргументом потрібно вказати файл, звідки потрібно проводити читання. Для читання символів з клавіатури вказується стандартний ідентифікатор **stdin** (**stdout** для **fputs()**).

Вивести значення рядка на екран можна за допомогою команд:

```
puts(<назва рядка>);
fputs(<назва рядка>, stdout);
printf("%s", <назва рядка>);
```

Приклади:

```
char line[81];
while(gets(line))
   puts(line);
```

```
char line[81];
while(fgets(line, 81, stdin))
   fputs(line, stdout);
```

Обидві наведені ділянки коду будуть виконуватися, поки не зустрінеться символ **EOF**, або поки програма не завершиться аварійно.

Крім того, стандарт C11 пропонує безпечні варіанти попередніх функцій: **scanf_s(), printf_s(), gets_s()**.

Посимвольно вводити чи виводити елементи рядка можна за допомогою команд циклу **for** або **while** і використанням функцій бібліотеки **<stdio.h>** **getchar()** та **putchar()**, чи їх файлових еквівалентів **getc()** та **putc()**. Наприклад:

```
char str[256];
int n;
for(n = 0; (str[n] = getchar()) != '\n'; ++n);
    str[n] = '\0';
for(n = 0; str[n]; ++n) putchar(str[n]);
```

Крім того, можна зустріти використання функцій **fgetc()** та **fputc()**. Вони еквівалентні до **getc()** та **putc()** і єдина різниця полягає в тому, що у деяких компіляторах останні можуть бути реалізовані як **#define** макроси.

### 5.2.2.Функції для опрацювання рядків

Для опрацювання масивів символів у мові С є стандартні функції, які описані у бібліотеці **<string.h>**. Розглянемо деякі з них [11]:

```
size_t strlen(const char *str);
```

повертає кількість символів у рядку.

```
char *strcat(char *dst, const char *src);
```

команда з'єднання рядків **dst** та **src**, результат записується в **dst,** повертає **dst**;

```
char *strncat(char *dst, const char *src, size_t num);
```

до змінної **dst** додає перших **num** символів рядка **src**;

```
char *strcpy(char *dst, const char *src);
```

копіює символи з рядка **src** в рядок **dst**;

```
char *strncpy(char *dst,const char *src, size_t num );
```

копіює перших **num** символів рядка **src** в рядок **dst**;

```
char *strchr(const char *str, int ch);
```

визначає перше входження деякого символу **ch** у рядок **str**, повертає рядок, який починається від першого входження заданого символу до кінця рядка;

```
char *strrchr(const char *str, int ch);
```

визначає останнє входження заданого символу **ch** у рядок **str**;

```
size_t strspn(const char *str1, const char *str2);
```

визначає номер першого символу, який входить у рядок **str1**, але не входить у рядок **str2**;

```
char *strstr(const char *str1, const char *str2 );
```

визначає в рядку **str1** підрядок, що починається з першого входження рядка **str2** у рядок **str1**;

```
char *strtok(char *str, const char *delimiters);
```

функція-токенайзер – розбиває рядок **str**, на окремі частини (лексеми), що розділені одним з символів-роздільників **delimiters**, якщо частина знайдена, повертає вказівник на неї, у іншому випадку повертає нуль. Функція зазвичай викликається кілька разів. При першому виклику **str** повинен відповідати потрібному рядку. При наступних викликах **str** повинен відповідати нулю, а пошук автоматично продовжуватиметься у початковому рядку. Нприклад:

```
char str[] ="- This, a sample string.";
char * pch;
printf ("Splitting string \"%s\"\n"
   "into tokens:\n",str);
pch = strtok (str," ,.-");
while (pch != NULL)
{
```

```
        printf ("%s\n", pch);
        pch = strtok (NULL, " ,.-");
    }
```

У результаті отримаємо:

```
    Splitting string "- This, a sample string."
    into tokens:
    This
    a
    sample
    string
```

Слід зауважити, що функція **strtok()** модифікує рядок – всі роздільники заміняються на нуль символи.

Наступні функції не є частиною стандартної бібліотеки і можуть бути відсутні у деяких компіляторах:

```
    char *strnset(char *str, int ch, size_t count);
```

вставляє **count** разів заданий символ **ch** перед рядком **str**;

```
    char *strupr(char *str);
```

перетворює усі малі літери рядка у великі;

```
    char *strlwr(char *str);
```

перетворює усі великі літери рядка у малі;

```
    char *strrev(char *str);
```

записує рядок у зворотному порядку.

Розглянемо результати застосування функцій до таких змінних:

```
        char lviv[] = "Lviv polytechnic",
            un[30] = "NU ",
            r1[30] = " ",
            r2[30] = "";
        char *p; int n;
```

| Застосування функцій | Результат |
|---|---|
| `n = strlen(lviv);` | `n = 16` |
| `strcat(un, lviv);` | `un = "NU Lviv polytechnic"` |
| `strncat(r1, lviv, 8);` | `r1 = " Lviv pol"` |
| `strcpy(r1, lviv);` | `r1 = "Lviv polytechnic"` |
| `strncpy(r2, lviv, 8);` | `r2 = "Lviv pol"` |
| `p = strchr(lviv, 'p');` | `p = "polytechnic"` |
| `p = strrchr(lviv, 'i');` | `p = "ic"` |
| `n = strspn(lviv, "Lviv");` | `n = 4` |
| `p = strstr(lviv, "tech");` | `p = "technic"` |
| `p = strtok(lviv, "iv");` | `p = "L"` |

У бібліотеці **<stdlib.h>** є стандартні функції перетворення типів даних. Зокрема, функція **atoi(r1)** перетворює рядок символів **r1** у дане цілого типу **int**, а функція **itoa(<числове дане>, r1, <система числення>)** – дане цілого типу **int** у рядок **r1**. Для перетворення даних типу **double** у рядок символів визначена функція **gcvt(<числове дане>, <кількість знаків у числі>, r1)**, а обернену дію виконує функція **strtod()**.

Розглянемо результати дії цих функцій для оголошених нижче змінних:

```
int n;
double f;
char r1[5], *p;
```

| Застосування функції | Результат |
|---|---|
| `n = atoi("12");` | `n = 12` |
| `itoa(12, r1, 10);` | `r1 = "12"` |
| `gcvt(-3.14, 4, r1);` | `r1 = "-3.14"` |
| `f = strtod("-3.1415", &p);` | `f = "-3.141500"` |

Рядки символів можна порівнювати між собою. Два рядки порівнюють зліва направо посимвольно, причому **'A' < 'B'**, **'B' < 'C'** тощо. "Більшим" вважається символ, який розміщений в алфавіті далі (він має більший номер у таблиці кодів ASCII). Для порівняння рядків у модулі **<string.h>** надана функція:

```
int strcmp(const char *str1, const char *str2);
```

порівнює рядки символів **str1** і **str2** з урахуванням регістра для латинського алфавіту. Результатом виконання є від'ємне число (якщо рядок **str1** менший від рядка **str2**), **0** (якщо рядки однакові) або додатне число (рядок **str1** більший за рядок **str2**).

Розглянемо результат дії функцій

| Функція | Результат |
|---|---|
| `n = strcmp("Spring", "spring");` | `n = -1` |
| `n = strcmp("spring", "Spring");` | `n = 1` |
| `n = strcmp("Spring", "Spring");` | `n = 0` |

Нехай задано рядок **"One two three"**. Визначити довжину рядка. Вивести на екран друге слово цього рядка:

```
#include <stdio.h>
#include <string.h>
int main()
{
    char r1[ ] = "One two three";
    printf("%s\n", r1);
    printf("The length of string = %d\n", strlen(r1));
```

```
        char *p= strtok(r1, " ");
        p = strtok(NULL, " ");
        printf("%s\n", p);
        return 0;
    }
```

Результат:

```
    One two three
    The length of string = 13
    two
```

Подамо ще один спосіб розв'язання задачі, в якому рядок розглядається як масив символів і функції для роботи з рядками не використовуються:

```
    #include <stdio.h>
    int main()
    {
        char r1[ ] = "One two three";
        printf("%s\n",r1);
        int len; for(len=0;r1[len];++len);
        printf("The length of string = %d\n",len);
        int n;
        for(n=0;r1[n]!=' ';++n);
        for(++n;r1[n]!=' ';++n)putchar(r1[n]);
        return 0;
    }
```

Результат:

```
    One two three
    The length of string = 13
    two
```

Нехай необхідно ввести п'ять назв столиць європейських країн та впорядкувати їх за алфавітом:

```
    #include <stdio.h>
    #include <string.h>
    int main()
    {
        char str[5][256];
        for(int i=0; i<5; ++i)
            scanf("%s", str[i]);
        for(int i=0; i<5; ++i)
            for(int j=0; j<4; ++j)
                if(strcmp(str[j], str[j+1]) > 0)
                {
```

```
                    char tmp[256];
                    strcpy(tmp, str[j]);
                    strcpy(str[j], str[j+1]);
                    strcpy(str[j+1], tmp);
                }
        for(int i=0; i<5; ++i) puts(str[i]);
        return 0;
    }
```

Результат:

| Вхідні дані | Вихідні дані |
|---|---|
| Kiev<br>Warsaw<br>London<br>Paris<br>Rome | Kiev<br>London<br>Paris<br>Rome<br>Warsaw |

Застосування функцій **getchar()** і **putchar()** ілюструється наступною програмою. Вона вводить символи із клавіатури і виводить їх у протилежному регістрі, тобто великі букви стають малими, і навпаки. Щоб зупинити виконання програми, необхідно ввести **Ctrl+Z**:

```
#include <stdio.h>
#include <ctype.h>
int main(){
    char ch;
    puts("Please input text. Press Ctrl+Z to finish");
    while((ch=getchar())!= EOF)
    {
        if(islower(ch))  ch = toupper(ch);
        else ch = tolower(ch);
        putchar(ch);
    }
    return 0;
}
```

Результат:

```
Please input text. Press Ctrl+Z to finish
abcdEFGH
ABCDefgh
^Z
```

У наведеній програмі для роботи з верхнім та нижнім регістром використовуються стандартні функції бібліотеки **<ctype.h>**: **islower()**, **toupper()** та **tolower()**. Їх назви говорять самі за себе.

Функція **getchar()** може породити кілька проблем. Звичайно, ця функція поміщає вхідні дані в буфер, поки не буде натиснута клавіша <**ENTER**>. Такий спосіб називається буферизованным вводом (*line-buffered input*). Для того щоб дані, що були введені, дійсно передалися програмі, необхідно натиснути клавішу <**ENTER**>. Крім того, при кожному виклику функція **getchar()** вводить символи по одному, послідовно розміщаючи їх у черзі. Якщо програма використовує інтерактивний діалог, таке гальмування стає дратівним фактором. Незважаючи на те що стандарт мови С дає змогу зробити функцію **getchar()** інтерактивною, ця можливість використовується рідко. Отже, якщо раптом виявиться, що наведена вище програма працює не так, як очікувалося, причина буде очевидна.

Функція **getchar()**, реалізована компілятором, може не відповідати вимогам інтерактивного середовища. У цьому випадку можна використовувати інші функції, що дають змогу зчитувати символи із клавіатури. У стандарті мови С не передбачено жодної функції, що гарантує інтерактивний ввід, але на практиці цей недолік заповнюється компіляторами.

Найбільш відомими альтернативами є функції **getch()** і **getche()**, бібліотеки **<conio.h>**. Їхні прототипи виглядають так.

```
int getch(void);
int getche(void);
```

У деяких компіляторах перед іменами цих функцій ставиться знак підкреслення. Наприклад, у компіляторі Microsoft Visual C++ ці функції називаються **_getch()** і **_getche()** відповідно.

Після натискання клавіші функція **getch()** негайно повертає результат, введений символ на екрані не відображається. Функція **getche()** аналогічна функції **getch()**, за одним виключенням: введений символ відображається на екрані (*get character echo*). В інтерактивних програмах замість функції **getchar()** найчастіше використовуються функції **getch()** і **getche()**. Однак, якщо компілятор не підтримує ці функції, або функція **getchar()**, реалізована ним, повністю відповідає вимогам інтерактивного середовища, варто застосовувати саме її.

Перепишемо попередню програму, заміняючи функцію **getchar()** функцією **getch()**. Також замінимо умову виходу з програми, оскільки тепер неможливо розпізнати комбінацію **Ctrl+Z**:

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>
int main()
{
```

```
    char ch;
    puts("Please input text. Press '.' to finish");
    while((ch=getch())!= '.')
    {
        if(islower(ch))  ch = toupper(ch);
        else ch = tolower(ch);
        putch(ch);
    }
    return 0;
}
```

Результат:

| Вхідні дані | Вихідні дані |
|---|---|
| **abcdEFGH.** | **ABCDefgh** |

У ході виконання цієї програми при кожному натисканні клавіші відповідний символ буде негайно переданий програмі і відображений на екрані. Ввід не буферизується.

Наступна програма демонструє застосування декількох основних консольних функцій вводу-виводу при роботі з комп'ютерним словником. Програма пропонує користувачеві ввести слово, а потім перевіряє, чи міститься це слово в списку відомих:

```
#include <stdio.h>
#include <string.h>
#include <conio.h>
int main()
{
    char  dic[][20] = {
       "car","ship","train","bike","plane"};
    char word[20], ch;
    do
    {
        puts("Please input a word");
        gets(word);
        int n;
        for(n=0; n<5; ++n)
        {
            if(!strcmp(dic[n], word))
            {
                puts("Word is found!");
                break;
            }
```

```
            }
            if(n==5)puts("There is no such word");
            puts("Continue? [y]/[n]");
        }while((ch = getch()) != 'n');
        return 0;
    }
```

Результати:

```
Please input a word
fox
There is no such word
Continue? [y]/[n]
Please input a word
ship
Word is found!
Continue? [y]/[n]
```

Нехай необхідно ввести текст, визначити, скільки разів в тексті зустрічається кожна буква латинського алфавіту:

```
#include <stdio.h>
#include <string.h>
int main()
{
    int letters[26], length = 0;
    char text[1024], ch;
    for(int i=0; i<26; ++i) letters[i] = 0;
    puts("Please input text. Press Ctrl+Z to stop");
    while((ch = getchar()) != EOF)
    {
        if((ch >= 'a')&&(ch <= 'z'))
            letters[ch - 'a']++;
        else if((ch >= 'A') && (ch <= 'Z'))
            letters[ch -'A']++;
        ++length;
    }
    puts("Letters frequency:");
    for(int i=0; i<26; i += 2)
        printf("%c: %7.3lf%%\t%c: %7.3lf%%\n",
            i + 'A', 100.0 * letters[i] / length,
            i+'A' + 1, 100.0 * letters[i+1] / length);
    return 0;
}
```

Результати:

```
Please input text. Press Ctrl+Z to stop
Time is money. (Edward George Bulwer-Lytton)
^Z
Letters frequency:
A:   2.222%     B:   2.222%
C:   0.000%     D:   4.444%
E:  13.333%     F:   0.000%
G:   4.444%     H:   0.000%
I:   4.444%     J:   0.000%
K:   0.000%     L:   4.444%
M:   4.444%     N:   4.444%
O:   6.667%     P:   0.000%
Q:   0.000%     R:   6.667%
S:   2.222%     T:   6.667%
U:   2.222%     V:   0.000%
W:   4.444%     X:   0.000%
Y:   4.444%     Z:   0.000%
```

При написанні програм звичайно необхідно робити різні маніпуляції з рядками: копіювати їх і переносити з одного місця пам'яті в інше, перевіряти наявність у рядках певної послідовності символів, об’єднувати, зменшувати рядки і т. п. Мова С містить багату колекцію функцій для роботи з рядками. Кілька загальних зауважень щодо використання функцій бібліотеки:

- якщо у функції виконується перенос символів рядка з одного місця в інше, то для рядка призначення потрібно попередньо зарезервувати місце в пам'яті;
- копіювання рядків з використанням неініціалізованого вказівника – одна з найпоширеніших помилок програмування, навіть у досвідчених програмістів;
- працюючи з рядками, необхідно обов'язково звертати увагу на попередження компілятора типу "*Підозріле перетворення вказівника*" (*Suspicious pointer conversion*) або "*Використання вказівника до ініціалізації*" (*Possible use of...before deffinition*). При виділенні місця для рядка-призначення варто передбачити місце для нуль-термінатора (`'\0'`).

### 5.2.3. Таблиця ASCII символів

Символи зберігаються в пам'яті як числові коди. Найбільш часто застосовується система кодування *ASCII* (*American Standard Code for Information Interchange*) – система кодів, у якій числа від 0 до 127 включно

поставлені у відповідність літерам, цифрам і символам пунктуації. Існують і інші розширені системи кодування. ASCII можна розглядати як 7-бітну схему кодування підмножини Unicode/UCS. UTF-8 сумісний з ASCII для кодів менше 128.

Мова С дає змогу представити більшість символів безпосередньо шляхом їх укладення в одинарні лапки, наприклад, **'A'** для символу **A**. Крім того, символ можна представити з використанням його вісімкового або шістнадцяткового коду, перед яким повинен знаходитися зворотній слеш, наприклад, **'\012'** та **'\0xa'** відповідають символу переходу на новий рядок **'\n'**. Керуючі послідовності подібного роду також можуть бути частиною рядка, скажімо, такий: **"Please \012welcome!"**.

Перші 32 символи задумані для керування периферійними пристоями і не призначені для відображення на екран (символ **^** відповідає клавіші **Ctrl**):

| Код | Назва | Керуюча послідовність | Символ | Опис |
|---|---|---|---|---|
| **0x00** | **NUL** | **\0** | **^@** | Null character |
| **0x01** | **SOH** | | **^A** | Start of Heading |
| **0x02** | **STX** | | **^B** | Start of Text |
| **0x03** | **ETX** | | **^C** | End of Text |
| **0x04** | **EOT** | | **^D** | End of Transmission |
| **0x05** | **ENQ** | | **^E** | Enquiry |
| **0x06** | **ACK** | | **^F** | Acknowledgment |
| **0x07** | **BEL** | **\a** | **^G** | Bell |
| **0x08** | **BS** | | **^H** | Back Space |
| **0x09** | **HT** | **\t** | **^I** | Horizontal Tab |
| **0x0a** | **LF** | **\n** | **^J** | Line Feed |
| **0x0b** | **VT** | **\v** | **^K** | Vertical Tab |
| **0x0c** | **FF** | **\f** | **^L** | Form Feed |
| **0x0d** | **CR** | **\r** | **^M** | Carriage Return |
| **0x0e** | **SO** | | **^N** | Shift Out / X-On |
| **0x0f** | **SI** | | **^O** | Shift In / X-Off |
| **0x10** | **DLE** | | **^P** | Data Line Escape |
| **0x11** | **DC1** | | **^Q** | Device Control 1 (XON) |
| **0x12** | **DC2** | | **^R** | Device Control 2 |
| **0x13** | **DC3** | | **^S** | Device Control 3 (XOFF) |
| **0x14** | **DC4** | | **^T** | Device Control 4 |
| **0x15** | **NAK** | | **^U** | Negative Acknowledgement |
| **0x16** | **SYN** | | **^V** | Synchronous Idle |
| **0x17** | **ETB** | | **^W** | End of Transmit Block |
| **0x18** | **CAN** | | **^X** | Cancel |

| Код | Назва | Керуюча послідовність | Символ | Опис |
|---|---|---|---|---|
| **0x19** | **EM** | | **^Y** | End of Medium |
| **0x1a** | **SUB** | | **^Z** | Substitute |
| **0x1b** | **ESC** | **\e** | **^[** | Escape |
| **0x1c** | **FS** | | **^** | File Separator |
| **0x1d** | **GS** | | **^]** | Group Separator |
| **0x1e** | **RS** | | **^^** | Record Separator |
| **0x1f** | **US** | | **^_** | Unit Separator |

Символи з 32-го по 126 є символами, що відображаються на екрані чи друкуються на принтері. Символ 32 це пробіл, символ 127 відповідає команді стирання DEL.

| | +0 | +1 | +2 | +3 | +4 | +5 | +6 | +7 |
|---|---|---|---|---|---|---|---|---|
| 0x20 | | ! | " | # | $ | % | & | ' |
| 0x28 | ( | ) | * | + | , | - | . | / |
| 0x30 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0x38 | 8 | 9 | : | ; | < | = | > | ? |
| 0x40 | @ | A | B | C | D | E | F | G |
| 0x48 | H | I | J | K | L | M | N | O |
| 0x50 | P | Q | R | S | T | U | V | W |
| 0x58 | X | Y | Z | [ | \ | ] | ^ | _ |
| 0x60 | ` | a | b | c | d | e | f | g |
| 0x68 | h | i | j | k | l | m | n | o |
| 0x70 | p | q | r | s | t | u | v | w |
| 0x78 | x | y | z | { | \| | } | ~ | |

### 5.3. Контрольні запитання

1. Яким чином відбувається оголошення та ініціалізація рядків символів (стрічок)?
2. Які функції для роботи з стрічками ви знаєте?
3. Назвіть операції порівняння стрічок, коротко поясніть результати їх дії.

### 5.4. Лабораторне завдання

1. Навчитися використовувати символьні масиви для розв’язання задач роботи зі стрічками.
2. Одержати індивідуальне завдання.
3. Побудувати блок-схеми алгоритмів відповідно до завдання.
4. Скласти програми на алгоритмічній мові С згідно завдання.
5. Відлагодити програми, виконати обчислення, проаналізувати отримані результати.

### 5.5. Зміст звіту

1. Титульний аркуш.
2. Мета роботи.
3. Індивідуальне завдання.
4. Блок-схема алгоритмів у відповідності до завдання.
5. Тексти програм у відповідності до завдання.
6. Результати обчислень.
7. Аналіз результатів, висновки.

### 5.6. Індивідуальні завдання

Завдання 1

Ввести з клавіатури своє прізвище, ім'я та по батькові як одне текстове дане. Виконати описані нижче дії. Результати вивести на екран.

1. Вивести ім'я та кількість букв у третьому слові.
2. Визначити скільки букв `'a'` у прізвищі.
3. Вивести ініціали.
4. Вивести прізвище та ініціали.
5. Вивести ім'я та кількість букв у прізвищі.
6. Визначити скільки букв `'o'` є в імені.
7. Вивести найдовше слово.
8. Вилучити усі букви `'a'` та `'o'` з прізвища.
9. Вивести ім'я у стовпчик.
10. Продублювати всі букви в імені.
11. Вивести ім'я у зворотному порядку.
12. Вивести прізвище у стовпчик у зворотньому порядку.
13. Вивести найкоротше слово.
14. Потроїти голосні букви у імені?
15. Вивести ініціали та прізвище у зворотньому порядку.
16. Вивести кількість букв у імені та прізвищі.

Завдання 2

Символьні рядки S1 та S2 довжиною до 100 символів вводити з клавіатури. Виконати описані нижче дії. Результати вивести на екран.

1. Підрахувати кількість приголосних у S1 та S2, що з обох боків оточені цифрами.
2. Дописати у S2 ті символи з S1, які не зустрічаються у S2.
3. Знайти у S1 найдовшу послідовність літер, що розташовані по зростанню, а в S2 – по спаданню (за алфавітом).
4. Порахувати кількість цифр з S2, які зустрічаються в S1.
5. Змінити порядок символів у S1 на зворотній. Замінити усі маленькі приголосні S2 на великі.
6. Видалити з S1 усі входження S2.
7. Записати, відокремивши пробілами, у середину S1 всі символи з S2, що не є буквами, або цифрами.
8. Відсортувати по зростанню S1 та по спаданню S2 (за алфавітом).
9. Вивести кількість приголосних з S1, які зустрічаються в S2.
10. Дописати у початок S1 символи, які є присутніми в обох рядках.
11. Розділити символи S1 на три групи: голосні, приголосні, цифри. Групи

розділити пробілами та відсортувати їх по зростанню (за алфавітом). Аналогічно для S2 – по спаданню.

12. Видалити усі пропуски з S1 та дописати їх у початок S2.
13. Підрахувати кількість входжень S2 у S1, та дописати їх у початок S2 в зворотньому порядку.
14. Підрахувати кількість голосних у S1, що з обох боків оточені приголосними. Аналогічно для S2 – кількість приголосних, оточених голосними.
15. Знайти у S1 найдовшу послідовність цифр, а в S2 – голосних літер.
16. Подвоїти пробіли між словами в S1 та потроїти в S2.

# ЛАБОРАТОРНА РОБОТА № 6.
# Вказівники в мові програмування С

## 6.1. Мета роботи

Навчитися використовувати вказівники при написанні програм на мові С.

## 6.2. Теоретичні відомості

### 6.2.1.Поняття вказівника

*Вказівник* (*pointer*) – особливий вид змінної, яка зберігає адресу об’єкту в пам'яті та таким чином вказує на нього.

Оголошення вказівника:

```
<тип> * <ідентифікатор>;
```

“*****” – астерікс (зірочка), декларує, що змінна є вказівником. Наприклад:

```
int a = 5; // a has type (int)
int *p     // p has type (int*)
```

Вказівники надають символічний спосіб роботи з адресами. Оскільки апаратні інструкції обчислювальних машин у великій мірі покладаються на адреси, вказівники дають змогу виражати дії в манері, близькій до машинного представлення. Така відповідність робить програми з вказівниками ефективними. Зокрема, вказівники пропонують ефективний метод маніпулювання масивами. Нагадаємо, що ім’я масиву це константний вказівник на його перший елемент. Вказівники відкривають широкі можливості для передачі аргументів у функції без необхідності їх копіювання.

### 6.2.2.Операції з вказівниками

Операція взяття адреси:

Операція, що нерозривно пов'язана з вказівниками – унарна операція *взяття адреси*: “**&**”. Наприклад:

```
p = &a;
```

Тут в **p** записується адреса змінної **a**, тобто **p** починає вказувати на **a**. Результатом застосування операції “**&**” є адреса об’єкту в пам'яті. Результат має тип “вказівник” на тип змінної. Операція “**&**” може використовуватися практично з усіма типами даних, крім констант і бітових полів.

Операція розіменування вказівника:

Щоб отримати об’єкт в пам’яті, на який вказує вказівник необхідно його розіменувати. Для цього існує *операція розіменування вказівника* “*****”. Наприклад:

```
int b = *p; // now b equals a
```

Не слід плутати оголошення вказівника, де астерікс декларує, що ідентифікатор

має тип “вказівник” і операцію розіменування, яка використовується у виразах. Результат операції – вміст комірки пам'яті, на яку вказує **p**. Слід зазначити, що зі змінною (виразом) ***p** можна працювати як зі звичайною змінною. Наприклад:

```
*p += 3; // now a equals 8
```

Операція присвоювання:

Операція *присвоювання* для вказівників аналогічна відповідній операції для інших типів даних. Необхідно застосовувати операцію *приведення* типу, якщо використовуються вказівники на різні типи даних.

Операція збільшення (зменшення) вказівника:

```
E + i;      E - i;
```

Тут **E** – змінна типу “вказівник”, а **i** – значення цілочисельного типу. Результат операції (**E+i**) – “вказівник”, що визначає адресу **i**-го елемента після даного, а (**E-i**) – на **i**-й елемент перед даним.

Операція складного присвоювання:

```
E += i;     E -= i;
```

Тут **E** – змінна типу “вказівник”, а **i** – значення цілочисельного типу. Ці операції аналогічні виразам (відповідно):

```
E = E + i; E = E - i;
```

Операції інкременту (декременту):

```
E++;        E--;        ++E;        --E;
```

Виконання даних операцій аналогічно відповідним операціям над цілочисельними типами, тобто вказівник буде зміщатися (збільшуватися або зменшуватися залежно від операції) на один елемент, фактично вказівник (адреса) зміниться на кількість байтів, що займає цей елемент у пам'яті.

Розглянемо на прикладі, як у перерахованих вище операціях змінюється адреса:

```
#include <stdio.h>
int main(){
    int   a, *pi = &a;
    float f, *pf = &f;
    printf("pi = %p  pf = %p\n", pi, pf);
    pi++;
    pf++;
    printf("pi = %p  pf = %p\n", pi, pf);
    return 0;
}
```

Можливий результат:

```
pi = 0x28ff14  pf = 0x28ff10
pi = 0x28ff18  pf = 0x28ff14
```

Насправді в результаті **pi++** вказівник зміниться на **sizeof(int)** байт, а при операції **pf++** – на **sizeof(float)** байт. Взагалі, якщо оголошено:

```
<тип> *p;
```

то операція **p = p + i**, де **i** – ціле, змінить **p** на **sizeof(<тип>)*(i** байт**)**.

Операція індексування:

```
E[i]
```

Тут **E** – змінна типу "вказівник", а **i** – значення цілочисельного типу. Ця операція повністю аналогічна виразу ***(E+i)**, тобто з пам'яті вибирається й використовується у виразі значення **i**-го елемента масиву, адреса якого присвоєна вказівнику **E.**

Згідно стандарту, постфіксна операція типу **E1[E2]** є еквівалентною до **(*((E1) + (E2)))** – компілятор завжди сам заміняє всі операції індексування на операції з вказівниками при генеруванні об'єктного коду. Тому слід зазначити, що наприклад звертання до елементів масиву **m[3]** і **3[m]** еквівалентні, хоча останній варіант рідко зустрічається на практиці.

Операція віднімання вказівників:

```
E1 - E2;
```

Тут **El, E2** – змінні типу "вказівник", причому вказівники на той самий набір даних і, природно, одного типу. Інакше операція безглузда. Тип результату залежить від компілятора і дорівнює кількості елементів, які можна розташувати в комірках пам'яті з **E2** по **E1.** Тип результату визначений в стандартній бібліотеці **<stddef.h>** як макрос **ptrdiff_t** і зазвичай це **int** або **long.**

Операції відношення:

```
E1 == E2;    E1 >= E2;    E1 > E2;
E1 < E2;     E1 <= E2;    E1 != E2;
```

Результат всіх операцій має тип **int**. Результат операції "**==**" буде дорівнювати одиниці (true), якщо **E1** і **E2** указують на той самий елемент в оперативній пам'яті. У протилежному випадку результат буде дорівнювати нулю (false). Результат операції "**>=**" одиниця (true), якщо об'єкт ***E1** розташований у пам'яті в старших адресах, ніж об'єкт ***E2**. Результатом операції "**<**" буде одиниця (true), якщо об'єкт ***E1** розташований в молодших адресах, ніж об'єкт ***E2.** Крім того, будь-який вказівник (адреса) може бути перевірений на рівність **(==)** або нерівність (**!=**) зі спеціальним значенням **NULL** (**NULL == 0**). Функції виділення пам'яті повертають **NULL** з появою яких-небудь помилок. Тому порівняння з **NULL** часто використовується для визначення помилок виділення пам'яті:

```
FILE *fp = fopen("FileName.txt", "r");
if(p == NULL) puts("File open error!");
```

### 6.2.3.Особливий тип вказівника – **void***

Вказівник типу **void*** визначає місце в оперативній пам'яті (адреса деякого байту), але не містить інформації про тип об'єкту. До використання значення, що перебуває по цій адресі, обов'язково повинна бути виконана операція приведення вказівника до деякого типу, тому що в протилежному випадку компілятору буде невідома довжина поля пам'яті, яке використовується в операції. До вказівника типу **void*** застосовуються наступні операції:

**=** – *просте присвоювання*;

**==, !=, >, <, <=, >=** – *операції порівняння*.

Приклад використання вказівника без типу (**void***) для демонстрації розміщення в пам'яті довгих даних (**long int**). При відображенні на екрані вміст байтів пам'яті виводиться в нормальній послідовності:

```
#include <stdio.h>
int main()
{
    long x = 0x12345678; // 78563412 in memory
    void *p = &x;
    // first byte
    printf("char  = %x\n",*(char*) p);
    // first two bytes
    printf("short = %x\n",*(short*)p);
    // all four bytes
    printf("long  = %x\n",*(long*) p);
    return 0;
}
```

Результат виконання, при умові що

**sizeof(char) = 1**,

**sizeof(short)= 2**,

**sizeof(long) = 4** (залежить від компілятора):

```
char  = 78
short = 5678
long  = 12345678
```

### 6.2.4.Вказівник на **char**

Оскільки мова С не підтримує елементи типу рядок, для роботи з ними зазвичай використовуються вказівники типу **char***. Якщо стрічкова константа використовується для ініціалізації вказівника типу **char***, то адреса першого символу змінної буде початковим значенням вказівника. Наприклад:

```
char *str = "cat";
```

Тут описується тільки вказівник **str**, і вказівник одержує початкове значення, рівне адресі першого елемента (символу **'c'**) стрічкової константи. Компілятор виділить пам'ять як для рядка (чотири байти), так і для розміщення значення вказівника.

Якщо стрічкова константа використовується в тих місцях виразу, де дозволяється застосовувати вказівник, компілятор підставляє у вираз замість константи адресу її першого символу. Наприклад:

```
char *str;
str = "cat";
```

Рядок можна ввести за допомогою функції **scanf**, використовуючи вказівник:

```
char *str, rt[20], s[15];
str = s;
scanf("%s", str);
scanf("%s", rt);
```

Ввести рядки можна й так:

```
scanf("%s", &str[0]);
scanf("%s", &rt[0]);
```

Також можна створити масив вказівників типу **char**:

```
char *strl[10];
```

Ініціалізацію масиву рядків і масиву вказівників можна виконати в такий спосіб:

```
char fg[][6] = {"one", "two", "three"};
char *strn[2];
strn[0] = "OMEGA";
strn[1] = "1234567";
char *strm[2] = {"DELTA","9876543"};
```

Ввести рядки можна наступними операторами:

```
char sd[4][30];
for(int i = 0; i < 4; i++)
    scanf("%s", sd[i]);
char *jk[4], s[4][30];
for(int i = 0; i < 4; i++){
    jk[i] = s[i];
    fgets(jk[i], 25, stdin);
}
```

Перевага використання масивів вказівників у тому, що можна оперувати не самими об'єктами, а тільки їхніми адресами, що дає значний виграш у швидкості виконання програми.

Як приклад роботи з рядками розглянемо програму, де сортується масив рядків, використовуючи вказівники на них. Ознакою кінця введення рядків буде введення порожнього рядка:

```
#include <stdio.h>
int main()
{
    char str[25][25], *ptrs[25];
    puts("Please input few words, "
         "input empty line to stop");
    int length;
    for(length = 0; ; ++length)
    {
        gets(str[length]);
        ptrs[length] = str[length];
        if(!str[length][0])break;
    }
    for(int j = 0; j < length; ++j)
        for(int g = 0; g < length - 1; ++g)
        {
            char *c1 = ptrs[g];
            char *c2 = ptrs[g + 1];
            int flag;
            for(; *c1 == *c2; ++c1, ++c2)
                if(!*c1)
                {
                    flag = 0;
                    break;
                }
            flag = *c1 - *c2;
            if(flag > 0)
            {
                char *p     = ptrs[g];
                ptrs[g]     = ptrs[g + 1];
                ptrs[g + 1] = p;
            }
        }
    puts("Sorted words:");
    for(int j = 0; j < length; ++j) puts(ptrs[j]);
    return 0;
}
```

Результати:

```
Please input few words, input empty line to stop
Kiev
Warsaw
London
Paris
Rome

Sorted words:
Kiev
London
Paris
Rome
Warsaw
```

При вирішенні завдання використовувався як двомірний масив, так і масив вказівників на рядки. Для того щоб поміняти рядки місцями, досить поміняти місцями вказівники на дані рядки в масиві вказівників. Масив вказівників сортується так, щоб перший елемент вказував на “наймолодший” за алфавітом рядок, а самий останній – на самий “найстарший” рядок. При введенні рядків кожний рядок порівнюється з нульовим для перевірки умови закінчення вводу.

Розглянемо приклад – обчислення довжини рядка:

```
#include <stdio.h>
#include <string.h>
int main()
{   char str[] = "1, 2, 3", *s;
    int l1 = 0; while(str[++l1]);
    int l2 = 0; for(s = str; *s++;); l2 = s-str-1;
    int l3 = strlen(str);
    printf("%d %d %d\n", l1, l2, l3);
    return 0;
}
```

Результати:

```
7 7 7
```

### 6.2.5.Зв'язок між вказівниками й масивами

У мові С при обробці елементів масиву зручно використовувати вказівник на цей масив. Будь-який доступ до елемента масиву, що здійснюється операцією індексування, може бути виконаний і за допомогою вказівника. Фактично, компілятор заміняє всі операції індексування на операції з вказівниками, при генеруванні об’єктного коду. Декларація **`int m[10];`**

визначає масив з десяти елементів, тобто блок з 10 послідовних об'єктів з іменами **m[0]**, **m[1]**, ..., **m[9]**. Запис **m[i]** є зміщенням адреси до і-го елемента масиву. Одночасно з виділенням пам'яті для десяти елементів типу **int** визначається значення константного вказівника **m**. Значення константного вказівника **m** дорівнює адресі елемента **m[0]**. Значення константного вказівника **m** змінити не можна, але його можна присвоїти іншому вказівнику й змінювати вже його значення, а також використовувати у виразах:

```
*(m + j + 2) = 5;      *(m + 3) = 1;
```

Якщо **pa** є вказівник на **int**, тобто визначений як:

```
int *pa;
```

то в результаті присвоювання:

```
pa = &m[0];
```

**pa** буде вказувати на нульовий елемент **m**; інакше кажучи, **pa** буде містити адресу елемента **m[0]**. Тепер присвоювання:

```
int x = *pa;
```

буде копіювати вміст **m[0]** у **x**. Якщо **pa** вказує на деякий елемент масиву, то **pa + 1** за визначенням вказує на наступний елемент, **pa + i** – на **i**-й елемент після **pa**, а **pa - i** – на **i**-й елемент перед **pa**.

Оскільки ім'я масиву є не що інше, як адреса його початкового елемента, то присвоювання **pa = &m[0];** можна реалізувати й так:

```
pa = m;
```

З іншого боку, якщо **pa** – вказівник, то у виразах його можна використовувати й з індексом, тобто **pa[i]**. Але між ім'ям масиву й вказівником, що має значення адреси масиву, є істотне розходження. Вказівник це змінна, тому можна записати вираз:

```
int m[5];
int *pa = m;
pa++;
```

А ім'я масиву – це вказівник-константа, тобто записи:

```
int x[5], *pa x, m[5];
m = pa; m++;            // Error
```

Для двовимірних масивів ім'я є вказівником-константою на масив вказівників-констант. Елементами масиву вказівників є вказівники-константи на початок кожного з рядків масиву: тому, при використанні вказівників "точкою відліку" може бути як найперший елемент масиву, так і перший елемент кожного з рядків, тобто можуть використовуватися як вказівник-константа, що задається ім'ям масиву, так і вказівники на рядки масиву. Наприклад, для массиву:

```
char str[3][50] = {"One", "Two", "Three"};
```

вираз:

```
putc(str[1][3]);
```

може бути записане у вигляді:

```
putc(*(str[1] + 3));
```

або:

```
putc(*(*(str + 1) + 3));
```

Якщо існують два вказівники **p1** і **p2**, що посилаються на той самий масив **mas**, то до них можна застосувати наступні операції:

```
p1++;
```

Установлює вказівник **p1** на наступний елемент масиву mas.

```
p2--;
```

Установлює вказівник **p2** на попередній елемент масиву mas.

```
p1+=i;
```

Установлює вказівник **p1** на і-й елемент після **p1**.

```
p2-=i;
```

Установлює вказівник **p2** на і-й елемент перед **p2**.

```
p2-p1;
```

Дає число елементів між **p1** і **p2**.

```
p1 == p2;
p1 != p2;
p1 < p2;
p1 > p2;
p1 <= p2;
p1 >= p2;
```

Виконують порівняння вказівників.

Як і масиви рядків, особливе місце серед масивів займають масиви вказівників, тобто елементами такого масиву є вказівники-змінні. Найбільш часто вони використовуються для компактного розташування в пам'яті рядків тексту, структурних змінних та інших великих об'єктів даних. Формат опису масиву вказівників має вигляд:

```
<тип> *<ім'я> [<розмір1>][<розмір2>]...;
```

Наприклад :

```
char *name[10];
```

Кожний елемент масиву **name** має тип **char*** (є вказівником на рядок). При визначенні масиву вказівників він може бути проініціалізований:

```
char *name[] = {"Alpha", "Beta", "Gamma"};
```

Компілятор резервує місце для трьох вказівників, кожний з яких одержує початкове значення, рівне адресі початку в пам'яті відповідного рядка. Існує велике розходження в розташуванні в пам'яті масиву вказівників і подібного йому двовимірного масиву типу **char**. Наприклад :

```
char names[][8] = { "Alpha", "Beta", "Gamma"};
```

У пам'яті масиви **name** і **names** можуть виглядати так (рис. 6.1).

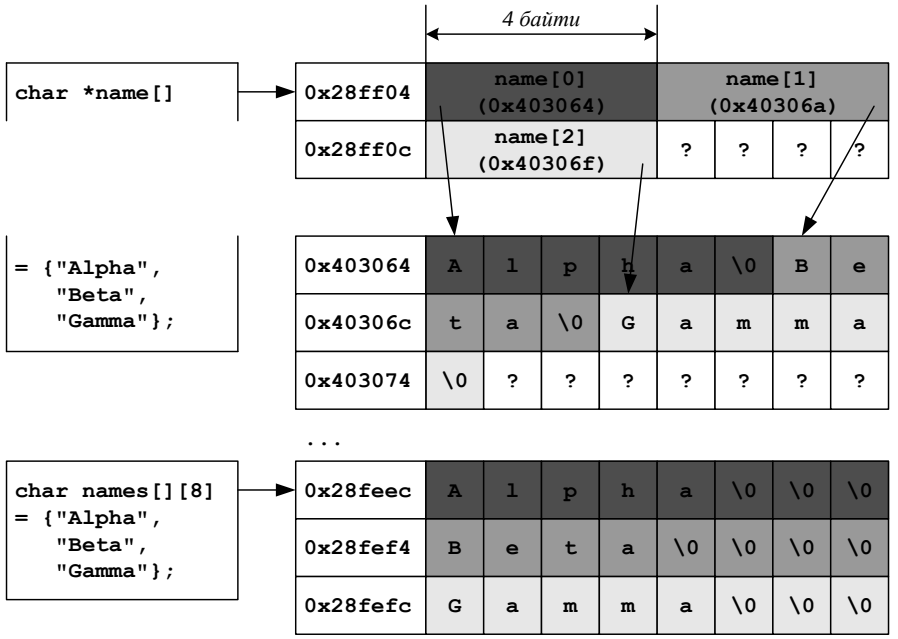



*Рис. 6.1 Розміщення рядків символів у пам'яті при ініціалізації*

Перевага використання вказівників у тому, що застосовуючи їх, можна маніпулювати не самими даними, а їхніми адресами, що дає виграш у швидкості виконання програми.

### 6.2.6. Використання вказівників при роботі з багатомірними масивами

При оголошенні масиву:

```
float fmas[2][3];
```

**fmas** – це вказівник-константа на матрицю. Вираз **frnas[0]** і **fmas[1]**, у свою чергу, також є вказівниками-константами на нульовий і перший рядки. Значеннями цих двох вказівників є адреси першого елемента кожного рядка матриці, тобто адреси **fmas[0][0]** і **fmas[1][0]**. Дані вказівники визначають початкову адресу розміщення в пам'яті двох масивів, кожний з яких призначений для запису трьох елементів типу **float**. Слід зазначити, що фактично в пам'яті ніякого масиву вказівників з елементами **fmas[0]** і **fmas[l]** компілятор не створює. Взаємозв'язок цих елементів можна зобразити в такий спосіб (рис. 6.2).

```
float fmas[2][3] = {{0.0, 0.1, 0.2},
                    {1.0, 1.1, 1.2}};
```

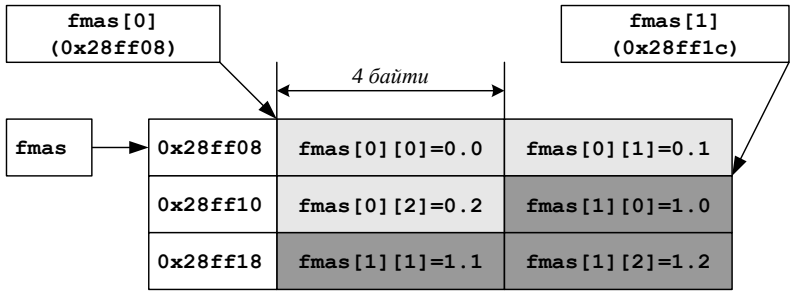



*Рис. 6.2 Розміщення в пам'яті двовимірного масиву елементів типу* **float**

Для установки вказівника на деякий рядок матриці використовуються вирази виду **fmas[i]** або ***(fmas+i)**. Вибірка самих значень елементів матриці реалізується виразами **fmas[i][j],** або ***(*(fmas+i)+j)**, або **(*(fmas+i))[j]**. Оскільки, елементи багатомірних масивів у пам'яті розміщаються послідовно, то можна звертатися до будь-якого елемента такого масиву, використовуючи вказівник і один індекс.

### 6.2.7.Вказівник на вказівник

У мові С можна оголошувати змінні, що мають тип "*вказівник на вказівник*". Ознакою такого оголошення є повторення астеріксу при оголошенні змінної. Число астеріксів визначає число "рівнів" вказівника. Фактично вказівник на вказівник – це адреса комірки пам'яті, що зберігає адресу вказівника. При визначенні вказівник на вказівник може ініціалізуватися. Наприклад:

```
int data     = 5;
int *ptr     = &data;
int **pptr   = &ptr;
int ***ppptr = &pptr;
```

Для отримання значення **data** всі нижченаведені записи рівносильні:

```
ptr[0]          *ptr                pptr[0][0]
**pptr          ppptr[0][0][0]      ***ppptr
```

Нехай оголошено масив:

```
int xi[] = {
    1,2,3,4,5,6,7};
int *pxi1[] = {
    xi,  xi+1, xi+2,
    xi+3, xi+4, xi+5,
    xi+6, xi+6, NULL};
int **pxi2 = pxi1;
```

Взаємозв'язок між цими змінними можна представити як (рис. 6.3).

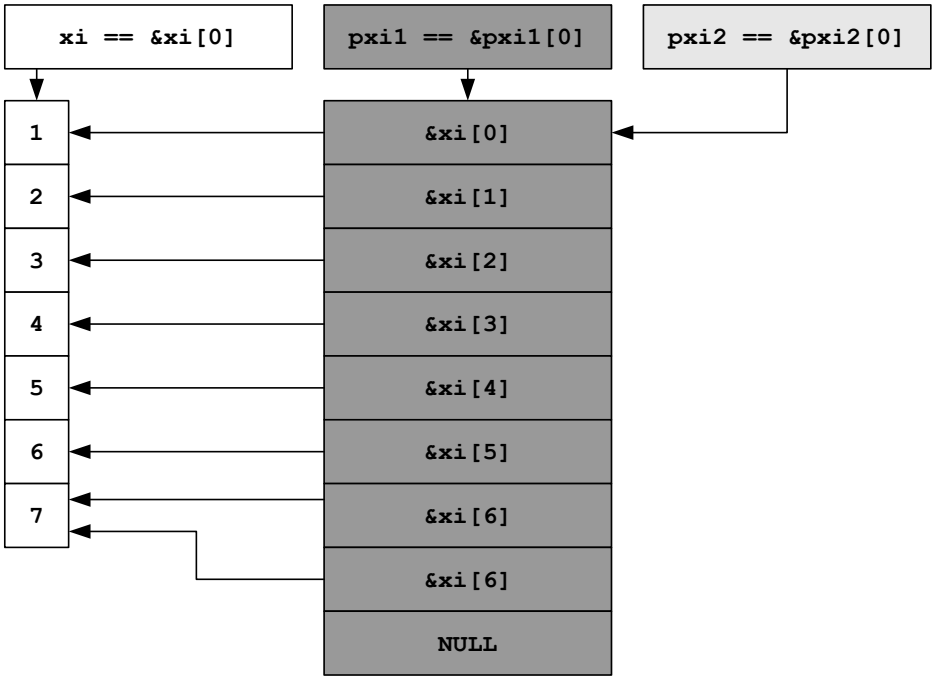



*Рис. 6.3 Розміщення рядків символів у пам'яті при ініціалізації*

Приклад звертання до елементів масиву з використанням вказівників:

```
#include <stdio.h>
int main()
{
    int x[]={1, 2, 3, 4, 5, 6},
        k;
    int *px1[]={
           x,  x+1, x+2, x+3, x+4,
           x+5,x+6, x+6, NULL},
        *ptr;
    int **px2 = px1;
    px2+=3;     k=**px2;            printf("%d ", k);
    ptr=*++px2; k= *ptr;            printf("%d ", k);
    px2=px1;    ptr=*++px2; k=*ptr; printf("%d ", k);
    ++*px2;     k=**px2;            printf("%d ", k);
    px2=px1;    k=*+++px2;          printf("%d ", k);
    return 0;
}
```

Результати виконання:

```
4 5 2 3 2
```

Приклад роздруку рядків масиву:

```
#include <stdio.h>
int main()
{
    char *str[] = {
        "first", "second", "third", "fourth", 0};
    char **pptr = str;
    while(*pptr)
        puts(*pptr++);
    return 0;
}
```

Результати виконання:

```
first
second
third
fourth
```

### 6.2.8.Приклади програм з використанням вказівників

З рядка виділити слова, записати їх у масив і вивести масив слів:

```
#include <stdio.h>
#include <conio.h>
int main()
{
    char x[100], y[20][100], *z;
    for(;;)
    {   int words = 0;
        puts("Press 'q' to exit, "
             "press 'a' to continue");
        switch(getch())
        {
            case 'q': return 0;
            case 'a':
                puts("Please input string");
                gets(x);
                z = x;
                for(words = 0; *z; ++words)
                {
                    int i;
                    for(i=0; *z != ' ' &&
                            *z; ++z, ++i)
                        y[words][i] = *z;
```

```
                    y[words][i] = '\0';
                    if(*z == ' ') ++z;
                }
                for(int i=0; i<words; ++i)
                    puts(y[i]);
                break;
            }
        }
        return 0;
    }
```

Результати виконання:

```
    Press 'q' to exit, press 'a' to continue
    Please input string
    Time is money
    Time
    is
    money
    Press 'q' to exit, press 'a' to continue
```

Використання складних оголошень:

```
    #include <stdio.h>
    int main()
    {
        int i;
        char *c[]={"1a","2b","3c","4d","5e"};
        char **cc[5];
        char ***ccc[5];
        char ****cccc[5];
        char *****ccccc[5];
        char ******cccccc[5];
        for(i=0;i<5;i++)
        {
            printf("c[%d]=%s ",i,c[i]);
            cc[i]=&c[i];
            ccc[i]=&cc[i];
            cccc[i]=&ccc[i];
            ccccc[i]=&cccc[i];
            cccccc[i]=&ccccc[i];
            printf("cc=%s "
                   "ccc=%s "
                   "cccc=%s "
                   "ccccc=%s "
```

```
                    "cccccc=%s\n",
                    *cc[i],
                    **ccc[i],
                    ***cccc[i],
                    ****ccccc[i],
                    *****cccccc[i]);
        }
        return 0;
    }
```

Результати виконання:

```
    c[0]=1a cc=1a ccc=1a cccc=1a ccccc=1a cccccc=1a
    c[1]=2b cc=2b ccc=2b cccc=2b ccccc=2b cccccc=2b
    c[2]=3c cc=3c ccc=3c cccc=3c ccccc=3c cccccc=3c
    c[3]=4d cc=4d ccc=4d cccc=4d ccccc=4d cccccc=4d
    c[4]=5e cc=5e ccc=5e cccc=5e ccccc=5e cccccc=5e
```

### 6.2.9.Динамічне виділення пам`яті

При вирішенні прикладних задач дуже часто виникає потреба працювати з об’єктами, що виникають та зникають у процесі виконання програми динамічно. Наприклад наперед не відомо скільки рядків тексту введе користувач в програму для їх деякого подальшого опрацювання. Можна виділити наперед деякий великий буфер пам’яті і працювати з ним. Альтернативним варіантом є використання засобів динамічного виділення пам’яті. При цьому, пам'ять виділяється з вільної області в міру потреби й повертається назад, тобто звільняється, коли необхідність у ній зникла. Область вільної пам'яті, доступної для виділення, перебуває між областю пам'яті, де розміщається код програми, і стеком, де розміщують всі локальні та глобальні змінні, значення констант, масиви тощо. Ця область називається *купою* (*heap*).

Динамічне виділення пам'яті в С забезпечується стандартними функціями бібліотеки **<stdlib.h>**, основна з яких:

```
    void* malloc(size_t size);
```

здійснює запит до операційної системи на виділення пам’яті розміром **size** байт. Якщо це вдається, то повертає **void**-вказівник на цю пам’ять. Якщо не вдається повертає **NULL**.

Коли потреба в пам'яті відпадає, її можна і потрібно звільнити за допомогою функції:

```
    void free(void *ptr);
```

Звільняє пам'ять, яка була попередньо виділена за допомогою **malloc()**, чи інших стандартних функцій, тобто повідомляє операційну систему, що дана пам'ять більше не використовується. Таким чином, цю пам'ять пізніше можна

виділяти знову. Якщо **ptr** не вказує на блок пам'яті, що був попередньо виділений (наприклад, він в силу деяких причнин вказує на якусь змінну чи нікуди не вказує) то поведінка функції не регламентується стандартом і може стати причною програмних помилок. Якщо **ptr** рівний **NULL** функція нічого не робить. Слід зауважити, що функція не змінює значення **ptr** – після її виконання, вказівник і далі буде вказувати на тепер вже недоступну пам'ять. Такі ситуації необхідно уважно відслідковувати! Не можна використовувати пам’ять, що не виділена – це причина більшості помилок і неправильної роботи програмних продуктів. Наприклад:

```
char *str = (char*)malloc(4 * sizeof(char));
str[0] = 'H'; str[1] = 'i'; str[2] = '!'; str[3] = 0;
putch(str); //Output "Hi!" message
free(str);
putch(str); //Error, str points to bad data
```

Приклад програми, що виводить стрічку випадкових символів (за допомогою функції бібліотеки **<stdlib.h>**: **rand()**), довжиною, яку вказує користувач. При цьому, пам’ять під стрічку виділяється динамічно:

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int n;
    char *buffer;
    printf("What length of the string "
           "do you want?\n");
    scanf("%d", &n);
    buffer = (char*)malloc((n+1) * sizeof(char));
    if(buffer == NULL)
        return -1;
    for (int i = 0; i < n; ++i)
        buffer[i] = rand() % 26 + 'a';
    buffer[n] = '\0';
    printf ("Random string: %s\n", buffer);
    free (buffer);
    return 0;
}
```

Результати:

```
What length of the string do you want?
15
Random string: nwlrbbmqbhcdarz
```

### 6.3. Контрольні запитання

1. Що таке вказівник?
2. Які оператори для роботи з вказівниками ви знаєте?
3. Який зв’язок між вказівником та масивом?

### 6.4. Лабораторне завдання

1. Навчитися використовувати вказівники при написанні програм на мові С.
2. Одержати індивідуальне завдання.
3. Побудувати блок-схеми алгоритмів відповідно до завдання.
4. Скласти програми на алгоритмічній мові С згідно завдання.
5. Відлагодити програми, виконати обчислення, проаналізувати отримані результати.

### 6.5. Зміст звіту

1. Титульний аркуш.
2. Мета роботи.
3. Індивідуальне завдання.
4. Блок-схема алгоритмів у відповідності до завдання.
5. Тексти програм у відповідності до завдання.
6. Результати обчислень.
7. Аналіз результатів, висновки.

### 6.6. Індивідуальні завдання

Завдання 1

З клавіатури вводиться динамічний рядок, виконати наведене нижче завдання. При доступі до елементів використовувати вказівники.

1. Рядок містить символи нуля та одиниці, підрахувати кількість нулів та одиниць.
2. Порахувати скільки разів входять літери `'P'` та `'S'`.
3. Рядок містить цілі числа, знаки арифметичної дії та дужки, перевірити, чи виконується баланс дужок `'('` та `')'`.
4. Рядок містить символи нуля та одиниці, перевірити, чи він симетричним відносно середини.
5. Перевірити, чи входить у нього цифри 5 та 7.
6. Порахувати кількість слів.
7. Перевірити, чи зустрічається у рядку подвоєння літер.
8. Порахувати кількість знаків пунктуації.
9. Порахувати скільки разів входить буквосполучення `"ED"` або `"ed"`.
10. Порахувати кількість слів, що починаються з літери `'a'`.
11. Визначити кількість входжень знаку `'\'` після появи знака `'#'`.
12. Перевірити, чи сьома з кінця літера рядка є літерою `'S'` або `'W'`.
13. Перевірити, чи входить літера `'S'` та через один символ є цифра `'1'`.
14. Перевірити, чи є слова, що починаються з `'a'` та закінчуються `'s'`.
15. Сформувати динамічний рядок, вивести його на друк, обчислити скільки разів у першу половину рядка входять літера `'S'`.
16. Перевірити, чи починаються слова нових речень з великої літери.

Завдання 2

Виконати наведене нижче завдання:

1. Задано текст. Створити масив вказівників на окремі абзаци. Посортувати їх за довжинами. Вивести посортовані абзаци на екран.
2. Задано текст. Створити масив вказівників на окремі речення. Посортувати їх за кодами літер. Вивести посортовані речення на екран.
3. Задано текст. Створити масив вказівників на окремі речення. Посортувати їх за довжинами. Вивести посортовані речення на екран.
4. Задано текст. Створити масив вказівників на окремі речення. Вивести речення у зворотньому порядку.
5. Задано текст. Створити масив вказівників на окремі речення. Посортувати кожне друге речення за кодами літер. Вивести посортовані речення на екран.

6. Задано речення. Створити масив вказівників на окремі слова. Посортувати їх за кодами літер. Вивести посортовані слова на екран.
7. Задано речення. Створити масив вказівників на окремі слова. Посортувати їх за довжинами. Вивести посортовані слова на екран.
8. Задано речення. Створити масив вказівників на окремі слова. Вивести слова у зворотньому порядку.
9. Задано речення. Створити масив вказівників на окремі слова. Посортувати кожне друге слово за кодами літер. Вивести посортовані слова на екран.
10. Задано слово. Створити масив вказівників на окремі літери. Посортувати їх за кодами літер. Вивести на екран.
11. Задано слово. Створити масив вказівників на окремі літери. Вивести у зворотньому порядку.
12. Задано слово. Створити масив вказівників на окремі літери. Посортувати кожну другу літеру за кодами літер. Вивести посортовані літери на екран.
13. Задано текст. Створити масив вказівників на окремі знаки пунктуації. Для кожного, вивести частину тексту від нього і до наступного.
14. Задано текст. Створити масив вказівників на окремі слова, що є числами. Посортувати їх за кодами за кодами літер. Вивести на екран посортовані числа.
15. Задано текст. Створити масив вказівників на окремі слова, що містять великі літери. Посортувати їх за кодами літер. Вивести на екран посортовані слова.
16. Задано текст. Створити масив вказівників на окремі слова, що містять тільки малі літери. Посортувати їх за кодами літер. Вивести на екран посортовані слова.

# ЛАБОРАТОРНА РОБОТА № 7.
# Підпрограми (функції) в мові програмування С

## 7.1. 1. Мета роботи

Ознайомитися із особливостями застосування функцій у мові С.

## 7.2. Теоретичні відомості

### 7.2.1.Підпрограми (функції)

Як ви збираєтеся організувати програму? Проектна філософія С передбачає використання функцій в якості будівельних блоків. Раніше були наведені приклади використання стандартної бібліотеки функцій мови С, коли застосовували такі функції, як **`printf()`**, **`scanf()`**, **`getchar()`**, **`putchar()`** і **`strlen()`**. Тепер можна перейти до більш активних дій – створення власних функцій.

*Функція* – це самодостатня одиниця коду програми, спроектована для виконання окремого завдання. Структура функції і способи її можливого використання визначаються синтакними правилами. У мові С функція відіграє ту ж саму роль, яку в інших мовах програмування відіграють функції, підпрограми та процедури, хоча деталі можуть відрізнятися. Деякі функції призводять до виконання дії, наприклад, функція **`printf()`** виводить дані на екран. Інші функції повертають значення, яке буде застосовуватися в програмі. Наприклад, функція **`strlen()`** повертає у місце свого виклику довжину зазначеного рядка. У загальному випадку функція може одночасно виконувати дії і повертати значення.

Функції позбавляють від необхідності в багаторазовому написанні одного і того ж коду. Якщо в програмі потрібно виконувати певну послідовність дій кілька разів, досить одного разу написати відповідну функцію. Потім цю функцію можна застосовувати всередині програми там, де вона необхідна, або ж використовувати її в різних програмах, подібно до того, як у багатьох програмах була задіяна функція **`putchar()`**. Крім того, навіть якщо завдання вирішується лише один раз єдиною програмою, використання функції має сенс, тому що це робить програму більш модульною, таким чином покращуючи її читабельність і спрощуючи внесення змін чи виправлень.

Багато програмістів вважають за краще думати про функції як про "*чорну скриньку*", визначену в термінах інформації, яка в неї надходить (її в*хідні параметри*), і значення або дії, які вона продукує (її *вихідні результати*). Програміста не турбує те, що відбувається всередині чорної скриньки, якщо тільки він сам не займаєтеся розробкою цієї функції. Наприклад, коли використовуєттся функція **`printf()`**, то відомо, що її вхідними параметрами є

керуючий рядок і можливо деякі аргументи. Також відомий результат, який функція **printf()** повинна згенерувати. Однак програміст зовсім не замислюється про код, що реалізовує **printf()**. Такий підхід щодо функцій дає змогу зосередитися на загальній структурі програми, не відволікаючись на деталі. До того, як приступати до написання коду, ретельно обміркуйте, що повинна робити функція і яка її роль в програмі.

### 7.2.2. Загальний вигляд функції

```
<тип>  <ідентифікатор>(<список формальних параметрів>)
{
    <оператор1>;
    <оператор2>;
    ...
}
```

**<тип>** – визначає тип значення, яке поверне функція у місце свого виклику використовуючи оператор **return.** Якщо тип не визначений, то функція поверне значення типу **int** по замовчуванню. При цьому, у більшості випадків компілятор видасть попередження типу:

```
warning: return type defaults to 'int'
```

оскільки не вказування типу повернення є застарілою практикою і починаючи з стандарту C99 строго не рекомендується.

Список *формальних параметрів* є списоком, розділених попарно комами, типів та імен змінних, значення яких отримує в якості *фактичних параметрів* функція при виклику. Подібно змінним, оголошеним всередині функції, формальні параметри є локальними змінними, що видимі для функції. Це означає, що можна не турбуватися, якщо їх імена будуть дублюватися в інших функціях. Значення цих змінних будуть присвоюватися при виконанні функції. Зверніть увагу, що стандарт мови С вимагає, щоб кожній змінній передував її тип. Тобто, на відміну від звичайних оголошень, не можна застосовувати список змінних, які мають один і той же тип. У випадку, коли функція немає параметрів, список параметрів є порожнім, але дужки не опускаються.

Для прикладу наведено правильний опис функції **foo**:

```
void foo( int x, int y, float z)
```

та опис, що є невірним, через **int x,y**:

```
void foo( int x, y, float z)
```

Інколи можна зустріти застарілі, так звані K&R (*Brian Kernighan ma Dennis Ritchie – автори мови С*) оголошення функцій:

```
void foo (x, y)
   int x;
   float z;
```

попри те, що починаючи з стандарту C99 такі оголошення є забороненими, деякі компілятори їх досі підтримують!

### 7.2.3. Вихід з функції

Існують два шляхи завершення виконання функції і повернення у програму, яка здійснила виклик. Перший шлях – послідовне проходження всього тіла функції, наприклад, функція, що друкує значення **x**:

```
void simple(int x)
{
    printf("%d", x);
}
```

Цей варіант допустимий тільки для функцій, що мають тип повернення **void**, тобто не повертають ніяких значень.

Другий шлях – завершення виконання функції з використанням оператора **return**. Наступна функція завершує виконання, якщо значення змінної **x** рівне нулю або досягнуто кінець функції. Оператор **return** викликає завершення функції, хоча вона не виконалася до кінця (якщо **x == 0**):

```
void divide(float x, float y)
{   float z;
    if(x == 0) return;
    z = x / y;
    printf("Result : %f", z);
}
```

### 7.2.4. Повернення значення

Для повернення значення з функції використовується оператор **return** із вказаним виразом, результат якого повертається. Наприклад:

```
int max(int a, int b)
{   int temp;
    if(a > b) temp = a;
    else temp = b;
    return temp;
}
```

Дозволяється використовувати більше одного оператора **return**. Це спрощує розуміння алгоритму. Наприклад, попередня функція **max()** може бути написана так:

```
int max(int a, int b)
{   if (a > b) return a;
    else return b;
}
```

Функції, які повертають значення можна використовувати так, як показано у прикладі:

```
if(max(x, y) > 20) printf("Greater\n");
```

Але функції не можна присвоювати значень (вона не є *l-значенням*):

```
max(x, y) = 20; //Error! function is not l-value
```

Часто виникать питання, чи потрібно оголошувати змінну для того, щоб повернути значення з функції. Відповідь – ні. Розглянемо приклад:

```
#include <stdio.h>
int mul(int a, int b)
{
    return a * b;
}
int main()
{
    int x, y;
    x = 10; y = 20;
    printf("%d", mul(x, y));
    return 0;
}
```

7.2.5.Прототип функції

У випадках, коли програма складається з кількох файлів, що підключаються директивами **#include**, або з об'єктних файлів, що містять вже скомпільований код деяких функції, їх оголошення необхідно проводити за допомогою *прототипів* – коротких декларацій, що вказують лише тип повернення та список типів формальних параметрів без вказання тіла функцій.

Прототип функції виконує два завдання. Перше полягає в ідентифікації типу значення, яке поверне функція, так, щоб компілятор міг згенерувати коректний код для типу даних, що повертаються. Друге завдання – у визначенні типу та кількості параметрів, які використовуються функцією. Загальний вигляд прототипу наступний:

```
<тип>  <ідентифікатор>(<список формальних параметрів>);
```

Прототипи звичайно розташовують перед головною програмою або у заголовочному файлі ( з розширенням **.h**). Приклад:

```
#include <stdio.h>
float sum( float, float); //Prototype
int main()
{
    float f, s;
    f = 10.5;
```

```
    s = 15.3;
    printf("%f", sum(f, s));
    return 0;
}
float sum(float a, float b)
{
    return a + b;
}
```

При передачі параметрів у функцію додатково виконуються приведення типів фактичних параметрів до типу, який вказано у прототипі функції.

Прототипи є потужним доповненням до мови. Вони дають змогу компілятору виявляти багато помилок або помилки, які могли бути допущені при використанні функцій. Якщо їх не виявити вчасно, вони перетворяться в проблеми, які можуть виявитися важкими для відстеження. Чи зобов'язаний програміст застосовувати прототипи? Ні – існує один спосіб уникнути прототипу, одночасно зберігши переваги прототипіювання. Причиною оголошення прототипу є повідомлення компілятору про те, як повинна використовуватися функція, до першого фактичного випадку її застосування. Того ж результату можна добитися, оголоствши повне визначення тіла функції до її першого використання. У цьому випадку визначення діє як власний прототип. Найчастіше це робиться з короткими функціями.

### 7.2.6. Функція типу **void**

У випадку, коли функція не повертає жодного значення, вона оголошується як **void**. Приклад правильного використання такої функції:

```
#include <stdio.h>
void sign(int);
int main()
{   sign(5);
    return 0;
}
void sign(int a)
{   if (a > 0) printf("Positive value\n");
    else printf("Negative value\n");
}
```

У випадку, коли у функції відсутні параметри, і вона не повертає значення, то її вигляд наступний:

```
#include <stdio.h>
void myname(void);
int main()
```

```
{   myname();
    return 0;
}
void myname(void)
{    printf("John");
}
```

### 7.2.7. Область видимості

Одна із сильних сторін мови С пов'язана з тим, що вона дає змогу управляти тонкими аспектами програми. Система управління пам'яттю в С служить ілюстрацією такого управління, дозволяючи визначати, яким функціям відомі ті чи інші змінні та наскільки довго змінна існує в програмі.

У кожному прикладі програми, що тут наводиться, дані зберігаються в пам'яті. Для цього існує апаратний аспект – будь-яке збережене значення знаходиться у фізичній пам'яті. У літературі по С для опису такої ділянки пам'яті застосовується термін *об'єкт*. Об'єкт може зберігати одне або декілька значень. У певний момент об'єкт може поки не містити збереженого значення, але він буде мати правильний розмір для відповідного значення.

Є також і програмний аспект – програмі потрібен якийсь спосіб доступу до об'єкта. Цього можна досягти, наприклад, шляхом оголошення змінної. Оголошення призводить до створення *ідентифікатора*. Ідентифікатор є ім'ям, що можна застосувати для позначення вмісту окремого об'єкта. Ім'я змінної – не єдиний метод позначення об'єкта. Наприклад, до нього можна звернутися за адресою через вказівник.

Ідентифікатор, що використовується для доступу до цього об'єкта, може бути описаний за допомогою його *області видимості*, що вказує, в якій частині програми цей ідентифікатор дозволено використовувати. Область видимості описує ділянку або ділянки програми, де можна звертатися до ідентифікатора. Змінна в С має одну з наступних областей видимості: в межах блоку, в межах функції, в межах прототипу функції і в межах файлу. *Блок* – це частина коду, що міститься між відкритою фігурною дужкою і відповідною їй закритою дужкою. Наприклад, блоком є все тіло функції. Будь-який складовий оператор всередині функції також вважається блоком. Змінна, визначена в блоці, має область видимості в межах блоку, і її видно від місця, де вона визначена, до кінця блоку, що містить визначення. Крім того, формальні параметри функції, хоча вони з'являються до відкритої фігурної дужки функції, мають область видимості в межах блоку і належать блоку, який містить тіло функції.

*Область видимості в межах функції* застосовується тільки до міток, що використовуються з операторами **goto**. Це означає, що якщо мітка вперше з'являється у внутрішньому блоці функції, її область видимості простягається на

всю функцію.

*Область видимості в межах прототипу функції* застосовується до імен змінних, що використовуються в прототипах функцій. Область видимості в межах прототипу функції поширюється від місця визначення змінної до кінця оголошення прототипу. Це означає, що при обробці аргументу прототипу функції компілятор цікавить тільки тип аргументу. Якщо вказані імена, то зазвичай вони не грають ніякої ролі і не обов'язково повинні збігатися з іменами, які застосовуються у визначенні функції.

Змінна, визначення якої знаходиться за рамками будь-якої функції, має *область видимості в межах файл*у. Змінну, що має область видимості в межах файлу, видно від місця її визначення і до кінця файлу, який містить її визначення[1].

### 7.2.8.Класи зберігання в пам’яті

Для зберігання даних в пам'яті мова С пропонує п'ять різних моделей, або *класів зберігання*. Щоб їх зрозуміти, спочатку розглянемо можливості зв’язування об’єктів та тривалості їх збереження.

Змінна в С має одне з трьох можливих *зв'язувань* (*linkage*): зовнішнє зв'язування, внутрішнє зв'язування або відсутність зв'язування. Змінні з областю видимості в межах блоку, функції або прототипу функції не мають зв'язування. Це означає, що вони є закритими для блоку, функції або прототипу, в якому визначені. Змінна з областю видимості в межах файлу може мати або внутрішнє, або зовнішнє зв'язування. Змінна з зовнішнім зв'язуванням може застосовуватися в будь-якому місці багатофайлового проекту програми, а змінна з внутрішнім зв'язуванням – де завгодно в межах файлу (одиниці трансляції). Наприклад:

```
int A = 5;          //file scope of visibility,
                    //external linkage
static int B = 3;   //file scope of visibility,
                    //internal linkage
int main ()
{/*...*/}
```

[1] Те, що для програміста виглядає як кілька файлів, для компілятора може виглядати як єдиний файл. Припустимо для прикладу, і це трапляється досить часто, що програміст включає один або більше бібліотечних файлів (з розширенням **.h**) в файл вихідного коду (з розширенням **.c**). У свою чергу, бібліотечний файл може включати інші заголовочні файли. В результаті можуть бути задіяні багато фізичних файлів. Однак препроцесор С по суті замінює директиву **#include** вмістом відповідного файлу. Таким чином, компілятор бачить єдиний файл, який містить інформацію з файлу вихідного коду і всіх заголовочних файлів. Такий файл називається *одиницею трансляції*. Коли описується змінна, що має область видимості в межах файлу, насправді вона буде видимою для всієї одиниці трансляції.

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

Об'єкт можна описати в термінах його *тривалості зберігання*, яка вказує, наскільки довго він залишається в пам'яті. Об'єкт в С має одну з чотирьох тривалостей зберігання: статичну, потокову, автоматичну або виділену. Якщо об'єкт має статичну тривалість зберігання, він існує прягом часу виконання програми. Змінні з областю видимості в межах файлу мають статичну тривалість зберігання. Зверніть увагу, що для змінних з областю видимості в межах файлу ключове слово **static** вказує тип зв'язування, а не тривалість зберігання. Потокова тривалість зберігання вступає в гру при паралельному програмуванні, коли виконання програми може бути розділене на кілька потоків. Об'єкт з потокової тривалістю зберігання існує з моменту його оголошення і до завершення потоку (ключове слово **_Thread_local**). Змінні з областю видимості в межах блоку зазвичай мають автоматичну тривалість зберігання, явно це можна вказати з допомогою ключового слова **auto**. Пам'ять для цих змінних виділяється, коли потік управління входить в блок, де вони визначені, і звільняється, коли потік управління залишає цей блок. Ідея полягає в тому, що пам'ять, яка використовується для автоматичних змінних, є робочим простором або тимчасової пам'яттю, яка може застосовуватися багаторазово. Наприклад, після завершення виклику функції пам'ять, яку вона використовувала для своїх змінних, може бути задіяна при виклику інших функцій.

У результаті утворюються п'ять класів зберігання:

| Клас зберігання | Тривалість зберігання | Область видимості | Зв'язування | Оголошення |
|---|---|---|---|---|
| Автоматичний | Автоматична | Блок | Немає | Блок, ключове слово **auto** |
| Регістровий | Автоматична | Блок | Немає | Блок, ключове слово **register** |
| Статичний з зовнішнім зв’язуванням | Статична | Файл | Зовнішнє | За межами всіх функцій, ключове слово **extern** |
| Статичний з внутрішнім зв’язуванням | Статична | Файл | Внутрішнє | За межами всіх функцій, ключове слово **static** |
| Статичний без зв’язування | Статична | Файл | Немає | Блок, ключове слово **static** |

Специфікатор **register** може використовуватися тільки зі змінними, що мають область видимості в межах блоку. Він поміщає[1] змінну в регістровий клас зберігання, що рівносильно запиту на мінімізацію часу доступу до неї. Він також запобігає взяття адреси цієї змінної.

### 7.2.9. Параметри та аргументи функцій

Виклик функції з передачею значення

Цей метод виклику функції полягає в тому, що значення аргументу *копіюється* як формальний параметр підпрограми. Тому зміни значення цього параметра всередині підпрограми не впливають на значення змінних, які використовувалися для виклику. Наприклад:

```
#include <stdio.h>
int sqr(int x);
int main()
{
    int t = 10;
    printf("%d %d", sqr(t), t);
    return 0;
}
int sqr(int x)
{
    x = x * x;
    return x;
}
```

У цьому прикладі значення аргументу для **sqr()** – 10, скопіювалося у параметр **x**. Коли відбулося присвоєння **x = x * x**, змінилося значення лише локальної змінної **x**. Змінна **t**, яка використовувалася для виклику **sqr()**, і надалі має значення 10. А на екран виведеться: **100 10**.

Виклик функції з передачею адрес змінних

Використовуючи вказівник на аргумент в якості формального параметра функції можна змінювати значення аргументу всередині функції. Вказівники передаються так само, як і інші типи. Звичайно, потрібно оголосити параметри як вказівник на тип. Попередній приклад з використанням адрес буде виглядати так:

```
#include <stdio.h>
int sqr(int*);
int main()
```

[1] В залежності від налаштувань рівня оптимізації коду для конкретного компілятора, він сам буде поміщати змінні в процесорні регістри мінімізуючи час доступу до них, тому використання цього специфікатора є в деякій мірі зайвим.

```
{
    int t = 10;
    printf("%d ", sqr(&t));
    printf("%d", t);
    return 0;
}
int sqr(int *x)
{
    *x = (*x) * (*x);  return *x;
}
```

У наведеному прикладі у функцію передається копія вказівника на комірку пам’яті (іншими словами, адреса в пам’яті). Ця адреса використовується для доступу до пам’яті. Вміст змінної **t** став рівним 100. На екрані: **100 100**

Виклик функцій з аргументом у вигляді масиву

Коли масив використовується як аргумент, передається лише адреса масива, а не копія всього масиву. Коли викликається функція з аргументом у вигляді масиву, передається лише адреса першого елемента масиву. Оскільки ім'я масиву – це адреса його першого елемента, фактичний параметр у вигляді імені масиву вимагає, щоб відповідний формальний параметр був вказівником. У цьому і тільки в цьому контексті С інтерпретує int **arr[]** як **int *arr**, тобто **arr** є типом вказівника на **int**. Оскільки в прототипах дозволено опускати ім'я, всі чотири наведених нижче прототипу еквівалентні:

```
int sum (int *arr, int n);
int sum (int *, int);
int sum (int arr[], int n);
int sum (int [], int);
```

У деклараціях функцій імена опускати не можна, тому такі дві форми визначення еквівалентні:

```
int sum (int *arr, int n)
{
    // ...
}
int sum (int arr[], int n)
{
    // ...
}
```

Допускається використовувати будь-який з чотирьох показаних вище прототипів з будь-яким з двох наведених оголошень.

Найбільш поширеною формою у написаних Сі-програмах передача масиву як аргумента є:

```
#include <stdio.h>
void display(int *num);
int main()
{
    int t[10];
    for(int i = 0; i < 10; i++) t[i] = i;
    display(t);
    return 0;
}
void display(int *num)
{   for(int i = 0; i < 10; i++)
        printf("%d", num[i]);
}
```

Функція із змінним числом параметрів

Мова програмування С дозволяє використання змінного числа аргументів. Наприклад функції **printf()** та **scanf()**. Ознакою функції з і змінним числом аргументів є три крапки (еліпсис) "**...**" у списку параметрів прототипу функції. Зустрівши "**...**", компілятор зупиняє контроль відповідності типів параметрів для такої функції. Звичайно, функція зі змінним числом параметрів повинна мати засіб визначення точного їх числа при кожному виклику. Далі наводяться два приклади функції **example()**, яка приймає довільне число аргументів і виводить на екран їх кількість та набуті ними значення. У першому варіанті функції число фактично переданих значень задає перший аргумент. У другому варіанті, ознакою кінця аргументів є передача нуля:

```
#include <stdio.h>
void example(int arg1, ...)
{ //pointer to first
  int *p = &arg1;
  printf("\nThere are %d "
    "arguments: ", arg1);
  for(; arg1; arg1--)
    printf("%d ", *(++p));
}
int main()
{ example( 1, 5);
  example( 2, 5, 6);
  example( 5, 5, 6, 7,8,9);
  return 0;
}
```

```
#include <stdio.h>
void example(int arg1, ...)
{ //pointer to first
  int *p = &arg1;
  printf("\nArguments: ");
  while(*p)
    printf("%d ", *p++);
  printf(", there are %d "
   "arguments", p-&arg1);
}
int main()
{ example( 5, 0);
  example( 5, 6, 0);
  example( 5, 6, 7, 8,9,0);
  return 0;
}
```

Результат виконання (перший і другий варіанти відповідно):

```
There are 1 arguments: 5
There are 2 arguments: 5 6
There are 5 arguments: 5 6 7 8 9

Arguments: 5 , there are 1 arguments
Arguments: 5 6 , there are 2 arguments
Arguments: 5 6 7 8 9 , there are 5 arguments
```

Ці програми будуть працювати тільки під 32-ми компіляторами. Зверніть увагу на те, що для доступу до фактичних параметрів функції **example()** використовується пересування вказівника вперед. Цей варіант відповідає порядку передачі параметрів функції через стек, прийтятому у мові С – останній параметр записується в стек самим першим[1]. Однак, більшість сучасних компіляторів мають функціональні можливості автоматично оптимізувати код програми, наприклад, передаючи параметри не через стек, а через процесорні кеші чи регістри. Така оптимізація унеможливлює використання коду наведених прикладів, оскільки невідомо де в пам'яті будуть знаходитися параметри функції і доступ до них черех вказівник буде неправильним.

У загальному випадку слід використовувати макроси для роботи з функціями зі змінним числом параметрів, описані в стандартній бібліотеці **<stdarg.h>**: **va_list**, **va_start()**, **va_arg()**, **va_end()**, що гарантують правильну роботу програм і переносимість коду.

**va_list** – спеціальний тип даних, що визначає вхідні параметри;

```
void va_start (va_list ap, paramN);
```

ініціалізує **ap** параметром **paramN**;

```
type va_arg (va_list ap, type);
```

дістає значення поточного аргументу **ap**, що має тип **type** і переміщає **ap** на наступний параметр;

```
void va_end (va_list ap);
```

завершує роботу з відповідними макросами. Цей макрос обов’язково повинен бути викликаний після кожного виклику **va_start()** до завершення виконання функції. Наприклад:

```
#include <stdio.h>
#include <stdarg.h>
void example(int arg1, ...)
{
```

[1] Порядок передачі параметрів можна вказати явно за допомогою нестандартних директив типу “**__cdecl**” (залежать від компілятора).

```
    va_list p;
    va_start(p, arg1);
    printf("\nThere are %d "
    "arguments: ", arg1);
    for(; arg1; arg1--) printf("%d ", va_arg(p,int));
    va_end(p);
}
int main()
{
    example( 1, 5);
    example( 2, 5, 6);
    example( 5, 5, 6, 7, 8, 9);
  return 0;
}
```

Результат виконання:

```
Arguments: 5 , there are 1 arguments
Arguments: 5 6 , there are 2 arguments
Arguments: 5 6 7 8 9 , there are 5 arguments
```

### 7.2.10. Рекурсія

У мові С функції можуть викликати самі себе. Функція називається рекурсивною, якщо вираз в тілі функції містить виклик самої функції. Для того, щоб алгоритмічна мова була рекурсивною, потрібно, щоб функції мали можливість викликати самі себе. Класичний приклад рекурсії демонструє функція **fact()**, яка обчислює факторіал цілого числа. Факторіал числа **n** є добутком всіх цілих чисел між **1** та **n**. Обидва варіанти цієї функції – рекурсивний та ітеративний наведені нижче:

```
int factRec(int n) // recursive
{
    if (n <= 1) return 1;
    return n * factRec(n-1);
}
int factDir(int n) // direct
{
    int d = 1;
    for(int i = 1; i <= n ; ++i) d *= i;
    return d;
}
```

Коли **factRec()** викликається з аргументом **1**, функція повертає **1**; в іншому випадку вона повертає добуток **n * factRec(n-1)**. Для обчислення цього виразу **factRec()** викликається з виразом **n-1**. Це повторюється до того часу, поки **n** не стане рівне **1**. При кожному виклику рекурсивної функції, її параметри копіюються в пам'яті (рис. 7.1).

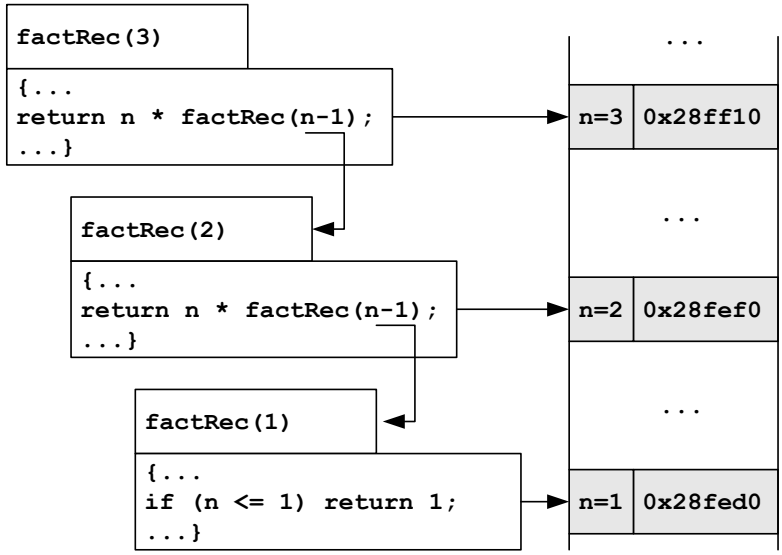



*Рис. 7.1 Виконання рекурсивної функції*

Необережне використання рекурсивних функцій може призвести до переповнення пам'яті (*stack overflow*) та аварійного завершення програми. При написанні рекурсивної функції, необхідно в її тілі забезпечити умову для нерекурсивного виходу. Якщо цього не зробити, то після виклику функції вона ніколи не поверне значення. Ця помилка є найбільш поширеною при написанні рекурсивних функцій.

### 7.2.11. Передача параметрів у функцію **main()**

Функція **main()** є першою функцією, що викликається при запуску будь-якої програми на мові С. Згідно зі стандартом, є два[1] можливі оголошення функції **main()**:

- без параметрів:

```
int main(void){ /*...*/ }
```

- з двома параметрами:

[1] Всі інші оголошення не є стандартними. Наприклад, існує оголошення з трьома параметрами, де третій параметр **char *env[]** є вказівником на масив параметрів середовища, де виконується програма, однак підтримка такого функціоналу залежиться від компілятора і системи.

```
int main(int argc, char *argv[]){ /*...*/ }
```

Зміст параметрів[1], які передаються функції **main()**:

- **argc** – число параметрів, що зберігаються в **argv**. Якщо оголошений, то обов'язково є більшим нуля;
- ***argv[]** – вказівник на масив стрічок з **argc** елементів (**argv[argc]** обов'язково рівний **NULL**). **argv[0]** – *назва програми* (і залежно від поточного каталогу, шлях до неї), якщо **argc>1**, то рядки з **argv[1]** по **argv[argc-1]** містять *параметри програми*, оголошені при її виклику з операційної системи, наприклад з командного раядка[2].

Наведемо приклад роботи функції **main()**:

```
#include <stdio.h>
int main(int argc, char *argv[])
{   if (argc != 3)
    {
        printf("Not enough parameters!\n");
        return 1; //Error
    }
    printf("Hello, %s %s\n", argv[1], argv[2]);
    return 0;
}
```

Наприклад, для лінійки операційної системи Windows. Якщо програма називається **helloName**, її можна викликати з командної стрічки:

```
D:\>cd D:\student\kn-1\jaworski\lab7
D:\student\kn-1\jaworski\lab7>helloName N. Jaworski
```

В результаті виконання програми на екрані з'явиться стрічка:

```
Hello, N. Jaworski
```

Наведемо ще один приклад (програма **args.exe**):

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

---

[1] Параметри функції **main()** можуть мати довільні імена, оскільки вони є локальними для цієї функції, але майже завжди вони відповідають стандартним **argc** та **argv**.

[2] Не всі середовища підтримують передачу параметрів в програму. Якщо це так, то згідно стандарту **argv[0][0]** буде містити нуль символ **'\0'**.

Результати:

```
D:\>cd D:\student\kn-1\jaworski\lab7
D:\student\kn-1\jaworski\lab7>args -r test [/b] end.
argc = 5
argv[0] = args
argv[1] = -r
argv[2] = test
argv[3] = [/b]
argv[4] = end.
```

Як і для звичайної функції повернення значення з **main()** виконується оператором **return**. Мова С допускає повторні виклики функції **main()** всередині програм.

### 7.2.12. Вказівники на функції

У мові С сама функція не може бути значенням змінної, але можна визначити вказівник на функцію. З ним можна оперувати, як зі змінною: передавати його іншим функціям, поміщати в масиви й т.д. Вказівник на функцію – такий тип, змінним якого можна присвоювати адреси точки входу у функцію, тобто адресу першої виконуваної команди. Ця змінна надалі може використовуватися для виклику функції замість її імені. Визначення вказівника на функцію має наступний загальний вид:

```
<тип>(* <назва>)(<список формальних параметрів>);
```

Наприклад:

```
double(*fp)(double); //function pointer
```

тут **fp** – вказівник на функцію, яка приймає значення типу **double** та повертає результат типу **double**. Перша пара дужок необхідна, без них:

```
double *fp(double);  //not a function pointer
```

означало б, що **fp** – функція, що повертає вказівник на **double**.

У наступній програмі виконується табулювання довільних функцій, що передаються через вказівник:

```
#include<stdio.h>
#include<math.h>
double myFun(double x)
{
    return sin(2 * x) * cos(x) + x * x - 7;
}
void tab(double (*fp)(double),
    double a, double b, int n)
{
    for(int i = 0; i < n; ++i)
```

```
            printf("%2d x=%6.3lf f(x)=%6.3lf\n",
                i + 1,
                a + (b - a) * i / (n - 1),
                fp(a + (b - a) * i / (n - 1)));
    }
    int main(void)
    {
        puts("sin(x):");   tab(sin, 0, 1, 5);
        puts("cos(x):");   tab(cos, 0, 1, 5);
        puts("myFun(x):"); tab(myFun, 0, 1, 5);
        return 0;
    }
```

Резултати:

```
    sin(x):
     1 x= 0.000 f(x)= 0.000
     2 x= 0.250 f(x)= 0.247
     3 x= 0.500 f(x)= 0.479
     4 x= 0.750 f(x)= 0.682
     5 x= 1.000 f(x)= 0.841
    cos(x):
     1 x= 0.000 f(x)= 1.000
     2 x= 0.250 f(x)= 0.969
     3 x= 0.500 f(x)= 0.878
     4 x= 0.750 f(x)= 0.732
     5 x= 1.000 f(x)= 0.540
    myFun(x):
     1 x= 0.000 f(x)=-7.000
     2 x= 0.250 f(x)=-6.473
     3 x= 0.500 f(x)=-6.012
     4 x= 0.750 f(x)=-5.708
     5 x= 1.000 f(x)=-5.509
```

## 7.3. Контрольні запитання

1. Коли виникає необхідність застосування функцій?
2. Який загальний вигляд функцій?
3. Як функція повертає значення?
4. Які завдання виконує прототип функції?
5. Як виглядає функція типу **`void`**?
6. Як реалізується виклик функції з передачею значень?
7. Як реалізується виклик функції з передачею адреси змінних?
8. Як реалізується виклик функції з масивом?
9. Як реалізується виклик функції зі змінним числом параметрів?
10. Що таке рекурсія?
11. Які Ви знаєте параметри функції **`main()`**?

## 7.4. Лабораторне завдання

1. Ознайомитися із особливостями застосування функцій у мові С.
2. Одержати індивідуальне завдання.
3. Побудувати блок-схеми алгоритмів відповідно до завдання.
4. Скласти програми на алгоритмічній мові С згідно завдання.
5. Відлагодити програми, виконати обчислення, проаналізувати отримані результати.

## 7.5. Зміст звіту

1. Титульний аркуш.
2. Мета роботи.
3. Індивідуальне завдання.
4. Блок-схема алгоритмів у відповідності до завдання.
5. Тексти програм у відповідності до завдання.
6. Результати обчислень.
7. Аналіз результатів, висновки.

### 7.6. Індивідуальні завдання

Завдання 1

Виконати завдання наведені нижче. Ввід-вивд даних та виконання інших окремих логічних дій необхідно реалізувати в окремих функціях. У головній функції необхідно виконувати лише їх виклик. Використання глобальних змінних не допускається. Інформація повинна передаватися у функції лише за допомогою параметрів.

1. Використовуючи метод половинного ділення знайти корінь рівняння: $2^x - x - 2 = 0$. Пошук здійснити на інтервалі [1,3]. Точність вводити з клавіатури.
2. Використовуючи метод лінійного пошуку знайти корінь рівняння: $x^2 - \sin 5x = 0$. Розв'язок лежить в інтервалі [0.4,0.8]. Точність вводити з клавіатури.
3. Використовуючи метод половинного ділення знайти корінь рівняння: $e^{2x} / \cos(2x) + 3x = 0$. Розв'язок лежить в інтервалі [–0.5,0.5]. Точність вводити з клавіатури.
4. Використовуючи метод лінійного пошуку знайти корінь рівняння: $\sin(3x + \pi/4)^3 - x^5 = 0$. Розв'язок лежить в інтервалі [–0.5,0.5]. Точність вводити з клавіатури.
5. Обчислити означений інтеграл функції: $1 + \cos^2 x$, на інтервалі $[0,\pi]$, використовуючи формулу лівих прямокутників. Крок вводити з клавіатури.
6. Обчислити означений інтеграл функції: $\sin^3 x - 3x$, на інтервалі $[0,\pi]$, використовуючи формулу правих прямокутників. Крок вводити з клавіатури.
7. Обчислити означений інтеграл функції: $7x^3 - x^2 + 3x + 2$, на інтервалі [2,3], використовуючи формулу середніх прямокутників. Крок вводити з клавіатури.
8. Обчислити означений інтеграл функції: $4x^5 - 3x^3 + x - 10$, на інтервалі [0,1], використовуючи формулу лівих прямокутників. Крок вводити з клавіатури.
9. Обчислити означений інтеграл функції: $\sin^3 x - x^4$, на інтервалі [1,6], використовуючи формулу правих прямокутників. Крок вводити з клавіатури.
10. Обчислити означений інтеграл функції: $\ln x / 5x^2 + 95x$, на інтервалі [0.1,0.2], використовуючи формулу середніх прямокутників. Крок вводити з клавіатури.
11. Обчислити суму $S(x) = \sum_{k=0}^{\infty} \frac{(-1)^{k+1} x^{2k-1}}{(2k-1)(2k+1)!}$ для значень $0 < x < 1$ з точністю, що задає користувач з клавіатури. На основі отриманих результатів обчислити означений інтеграл цієї функції. Крок вводити з клавіатури.

12. Обчислити суму $S(x)=\sum_{k=0}^{\infty}\frac{x^{2k+1}}{(2k+1)!}$. для значень 0 < *x* < 1 з точністю, що задає користувач з клавіатури. На основі отриманих результатів обчислити означений інтеграл цієї функції. Крок вводити з клавіатури.
13. Обчислити суму $S(x)=\sum_{k=1}^{\infty}(-1)^{k+1}\frac{x^{2k}}{2k(2k-1)}$. для значень 0 < *x* < 1 з точністю, що задає користувач з клавіатури. На основі отриманих результатів обчислити означений інтеграл цієї функції. Крок вводити з клавіатури.
14. Обчислити суму $S(x)=\sum_{k=0}^{\infty}\frac{\cos(k\pi/4)}{k!}x^k$. для значень 0 < *x* < 1 з точністю, що задає користувач з клавіатури. На основі отриманих результатів обчислити означений інтеграл цієї функції. Крок вводити з клавіатури.
15. Обчислити добуток $P(x)=\prod_{n=3}^{\infty}\frac{(n+2)!+nx}{(2n)!}$ для значень 0 < *x* < 1 з точністю, що задає користувач з клавіатури. На основі отриманих результатів обчислити означений інтеграл цієї функції. Крок вводити з клавіатури.
16. Обчислити добуток $P(x)=\prod_{k=2}^{\infty}(-1)^{k+1}\frac{x^{2k-1}+\ln x}{2xk(k+2)!}$ для значень 0 < *x* < 1 з точністю, що задає користувач з клавіатури. На основі отриманих результатів обчислити означений інтеграл цієї функції. Крок вводити з клавіатури.

Завдання 2

Виконати аналіз текстового файлу (текст довільний). Ввід-вивд даних та виконання інших окремих логічних дій необхідно реалізувати в окремих функціях. У головній функції необхідно виконувати лише їх виклик. Використання глобальних змінних не допускається. Інформація повинна передаватися у функції лише за допомогою параметрів. Назва текстового файлу та інші вхідні дані передаються в програму через аргументи функції **`main()`**. Вихідні дані виводяться на консоль.

1. Обчислити частоту розділових знаків (для кожного кількість та %).
2. Обчислити частоту малих літер (для кожної кількість та %).
3. Обчислити частоту великих літер (для кожної кількість та %).
4. Обчислити частоту цифр (для кожної кількість та %).
5. Визначити кількість знаків з пробілами і без пробілів.
6. Визначити кількість слів.
7. Визначити кількість рядків та абзаців.

8. Визначити, скільки разів входить задане користувачем слово.
9. Визначити довжину найдовшого слова кожного рядка, вивести слова та їх довжини.
10. Визначити довжину найкоротшого слова кожного рядка, вивести слова та їх довжини.
11. Обчислити частоту слів (для кожного кількість та %).
12. Обчислити кількість слів по абзацах.
13. Вивести всі слова, що починаються з літер, які вказані користувачем (слова не дублювати, вивести кожне окреме слово та кількість його входжень).
14. Поміняти місцями перший та другий абзаци.
15. Замінити усі входження слова, вказаного користувачем на інше слово, що також вказане користувачем.
16. Вивести всі слова тексту відсортовані за алфавітом (слова не дублювати, вивести кожне окреме слово та кількість його входжень).

# ЛАБОРАТОРНА РОБОТА № 8.
# Структури та об’єднання в мові програмування С

## 8.1. Мета роботи

Навчитися використовувати структури та об’єднання у мові С.

## 8.2. Теоретичні відомості

### 8.2.1.Структури

Оголошення структур

Одним з найбільш важливих кроків при проектуванні програми є вибір відповідного способу представлення даних. У багатьох випадках простих змінних або навіть масиву виявляється недостатньо. Мова С дозволяє розширити можливості представлення даних за допомогою змінних складних типів, тобто *змінних типу структура*. У своїй базовій формі струкри С є досить гнучким засобом, щоб представляти широке розмаїття даних, а також вона дає змогу винаходити нові форми. *Структура* – це набір з однієї або більше змінних, можливо різних типів, згрупованих під одним ім'ям для зручності обробки. Оголошення структури є генеральним планом, який описує спосіб формування структури. Іноді оголошення структури називають *шаблоном*:

```
struct <ідентифікатор>
{
    <тип> <ідентифікатор поля1>;
    <тип> <ідентифікатор поля2>;
    ...
};
```

Поняття структура застосовується в двох сенсах. Одним з них є шаблон, що повідомляє компілятору, як представляти дані, але не призводить до виділення простору в пам'яті для цих даних. Наступний крок полягає в створенні змінних типу структури, і в цьому полягає другий сенс поняття. Рядок програми, який створює змінну типу структури, має вигляд:

```
struct <назва структури> <змінна1>, <змінна2>, ...;
```

Змінні можна також оголошувати при створенні шаблону структури, після закритої фігурної дужки.

Розглянемо приклад:

```
struct DATE
{
    int  day;
    int  year;
    char monName[4];
} d;
```

Опис структури, що складається з взятого в фігурні дужки списку полів, починається з ключового слова **struct**. Після слова **struct** пишеться ідентифікатор, так звана, *мітка* структури (тут це **DATE**). Ідентифікатор може бути відсутнім. Така мітка іменує структури цього виду і може використовуватися надалі як скорочений запис шаблону. Елементи або змінні, згадані в структурі, називаються *полями* (елементами, компонентами). Область видимості ідентифікаторів полів є в межах блоку шаблону структури. Оголошення **struct DATE d;** визначає змінну **d** як структуру типу **DATE.**

Структура не може містити в собі змінні свого типу, тобто наступне оголошення шаблону буде неправильним:

```
struct man
{
    char *fam;
    char *name;
    struct man any;    //Error!
};
```

Однак структура може включати елементи, що є вказівниками на оголошений тип цієї ж структури:

```
struct man
{
    char *fam;
    char *name;
    struct man *pMan;   //Ok
};
```

Доступ до полів структури

Операція членства структури "**.**" зв'язує ім'я структури та ім'я члена. Вона називається *селектором*:

```
<змінна типу структура>.<поле відповідної структури>;
```

Наприклад:

```
   d.day     = 1;
   d.year    = 2017;
   d.monName = "Jan";
```

В програмі це може бути реалізовано наступним чином:

```
#include <stdio.h>
#include <string.h>
struct DATE
{   int  day;
    int  year;
    char monName[4];
};
```

```
int main(void)
{
    struct DATE d;
    d.day = 1;
    d.year = 2017;
    strcpy(d.monName, "Jan");
    printf("Date - %d %s %d\n",
        d.day, d.monName, d.year);
    return 0;
}
```

Результатом виконання програми буде напис : **Date - 1 Jan 2017**.

Структури можуть бути вкладеними, наприклад:

```
struct STUDENT
{   char        name[25];
    char        address[40];
    long int    zipCode;
    long int    stNumber;
    double      scholarship;
    int         course;
    char        group[7];
    struct DATE birthDate;
    struct DATE hereDate;
};
```

Структура **STUDENT** містить дві структури типу **DATE** . Якщо змінна **person** визначається як **struct STUDENT person**, то запис:

```
person.birthdate.year
```

буде посилатися на рік народження студента.

Доступ до полів структури через вказівник

Над змінною структури можна виконати операції взяття адреси змінної та присвоєння її вказівнику:

```
struct DATE d, *p;
p = &d;
```

Для демонстрації прикладу використання цих операцій над структурами перепишемо наведену вище програму так, щоб ввід даних про дату відбувався в фунгції **inputDate()**, а вивід в процедурі **printDate()**. В такому випадку програма набире наступного вигляду:

```
#include <stdio.h>
#include <string.h>
struct DATE
{
```

```
    int  day;
    int  year;
    char monName[4];
};
struct DATE inputDate(
    int day, int year, char *month)
{
    struct DATE tmp;
    tmp.day  = day;
    tmp.year = year;
    strcpy(tmp.monName, month);
    return tmp;
}
void printDate(struct DATE *d)
{
    printf("Date - %d %s %d\n",
        d->day, d->monName, d->year);
}
int main(void)
{
    printDate(&inputDate(1, 2017,"Jan"));
    return 0;
}
```

Розглянемо детальніше функцію **void printDate(struct DATE * d)**. Як видно з оголошення функції, у якості формального параметра використано вказівник на структуру. В такому випадку для звертання до полів структури використовується селектор вказівника "**->**":

```
<вказівник типу структура> -> <поле структури>;
(*<вказівник типу структура>).<поле структури>;
```

Наприклад:

```
struct DATE d;

d.date = 1;
d.year = 2017;
strcpy(d.monName,
       "Jan");
```

```
struct DATE d,*d1;
d1 = &d;
d1->date = 1;
d1->year = 2017;
strcpy(d1->monName,
       "Jan");
```

Масиви структур

Структури можна об'єднувати в масиви. Тоді запис **struct DATE d[5];** відображає масив з п'яти елементів типу **DATE**. Доступ до полів структури відбувається наступним чином:

```
d[0].year;
d[1].date;
```

Розмір структури можна взнати використавши стандартну операцію. У мові С передбачена унарна операція **sizeof()**, яка виконується під час компіляції, дозволяючи обчислити розмір будь-якого об'єкта. Вираз:

```
sizeof(object);
```

повертає ціле число, що дорівнює розміру вказаного об'єкту[1]. Об'єкт може бути фактичною змінною, масивом і структурою, або іменем основного типу, як **int** або **double**, або іменем похідного типу, як структура. Наприклад:

```
int size;
struct DATE d;
size = sizeof(d);
```

Ініціалізація структур

При визначенні структурних змінних можна ініціалізувати їхні поля. Ця можливість подібна ініціалізації масиву й слідує тим же правилам:

```
<ім'я шаблону> <ім'я змінної структури> = {
    <значення1>, <значення2>, ...
};
```

Компілятор присвоює **<значення1>** першій змінній в структурі, **<значення2>** – другій змінній й т.д. Тут необхідно додержуватися деяких правил:

- значення, що присвоюються, повинні співпадати по типу з відповідними полями структури;
- можна оголошувати кількість значень, що присвоюються, меншу, ніж кілкість полів. Компілятор присвоїть нулі іншим полям структури;
- список ініціалізації послідовно присвоює значення полям структури, вкладених структур і масивів.

Наприклад:

```
struct date{int day, month, year;}d[5] = {
    {1, 3, 1980},
    {5, 1, 1990},
    {1, 1, 1983}};
```

Проініціалізовані перші три елементи массиву.

---

[1] Розмір визначається в байтах, що залежно від архітектури комп'ютера можуть містити вісім або більше біт. Результатом **sizeof()** є ціле число типу **size_t**, якє згідно зі стандартом може зберігати максимальний розмір теоретично допустимого об'єкту будь-якого типу, у тому числі і масиву. **sizeof(char)**, **sizeof(signed char)** та **sizeof(unsigned char)** завжди повертають **1**.

### 8.2.2.Об'єднання

*Об'єднання* подібне структурі, але в кожен момент часу може використовуватись (або є активним) тільки один з його компонентів. Тип об'єднання може задаватися записом виду:

```
union <МІТКА>
{
    <тип> <компонент1>;
    <тип> <компонент1>;
    ...
};
```

Для кожного з цих компонентів виділяється одна і та ж область пам'яті, тобто вони перекриваються. Хоча доступ до цієї області пам'яті можливий через використання будь-якого з компонентів. Компонент для цієї мети повинен вибиратися так, щоб отриманий результат не був беззмістовним. Доступ до компонент об'єднання відбувається тим самим способом, що і для структур.

Об'єднання застосовуються:

- для мінімізації об'єму пам'яті, що використовується, якщо в кожний момент часу тільки один об'єкт з багатьох є активним;
- для інтерпретації основного представлення об'єкта одного типу, так ніби цьому об'єкту був присвоєний інший тип.

В якості прикладу визначення об'єкта типу **union** розглянемо об'єднання **shape**, яке визначається наступним чином:

```
struct position
{   int x; //sizeof(int) bytes
    int y; //sizeof(int) bytes
};
union
{
    float radius; //circle sizeof(float) bytes
    float a[2];   //rectangle sizeof(float)*2 bytes
    int b[3];     //triangle sizeof(float)*3 bytes
    struct position p;//coordinate sizeof(int)*2 bytes
} shape;
```

Розмір об'єднання **shape** дорівнює не суммі всіх байтів, як у випадку структури, а розміру найбільшого поля (компоненти). В цьому прикладі є сенс використовувати тільки той компонент, який отримав останнім своє значення.

Стандарт С11 дає можливість використовувати у якості полів анонімні об'єднання та структури. *Анонімне об'єднання* (*або структура*) – це неіменоване об'єднання (або структура), що є членом структури або об'єднання.

Наприклад:

```
struct names
{
    char first[20];
    char last[20];
};
struct person
{
    int id;
    struct names name;
};
struct person ted = {8483, {"Ted", "Grass"}};
```

Стандарт С11 дає змогу визначати структуру **person**, використовуючи в якості члена вкладену неіменовану структуру:

```
struct person
{
    int id;
    struct{char first[20]; char last[20];};
};
```

Цю структуру можна було б ініціювати в тій же манері:

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

Але доступ до членів спрощується, оскільки для цього застосовуються імена членів як **first**, так, ніби вони були членами **person**:

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

Зрозуміло, можна було б просто зробити **first** і **last** безпосередніми членами структури **person**, позбувшись від вкладеної структури. Засіб анонімності є більш корисний з вкладеними об'єднаннями.

### 8.2.3.Вирівнювання

Кожен тип об'єкту має властивість, що називається *вирівнюванням*. Вирівнювання рівне цілому значенню типу **size_t** (зазвичай **unsigned long**) та представляє число байтів між послідовними адресами, що можуть бути виділені для об'єкту даного типу. Згідно стандарту та апаратної реалізації оперативної пам’яті комп’ютерів, допустимі значення вирівнювання є цілими невід'ємними степенями двійки. Для того, щоб задовольнити вимоги до вирівнювання всіх членів структури, після деяких її членів компілятор може автоматично вставити відступи. Це обов’язково слід враховувати при використанні структур та об’єднань.

З появою стандарту С11 з’явилися стандартні засоби мови С для контролю вирівнювання даних. Зокрема оператор часу компіляції:

```
_Alignof( type-name );
```

Повертає вирівнювання типу **type-name**. Якщо **type-name** являє собою масив, результат є вирівнюванням елемента масиву. **type-name** не може бути функцією або неповним типом. Підключивши стандартну бібліотеку **<stdalign.h>**, цей оператор можна замінити на більш звичний **alignof()**.

Оператор часу компіляції:

```
_Alignas ( expression );
_Alignas ( type );
```

що може використовуватися тільки при оголошенні об'єктів. Він не може бути використаний в оголошеннях параметрів функції. При використанні в оголошеннях, заявлений об'єкт буде мати вирівнювання рівне:

- результату виразу, якщо він не дорівнює нулю і є додатнім степенем двійки;
- вирівнювання типу, тобто, до **alignof(type)** ;

Якщо вираз має значення нуль, то специфікатор не має ніякого ефекту. Підключивши стандартну бібліотеку **<stdalign.h>**, цей оператор можна замінити на більш звичний **alignas()**. Приклади:

```
#include <stdalign.h>
#include <stdio.h>
struct sse_t
{
    alignas(16) float sse_data[4];
};
struct data
{
    char x;
    alignas(128) char cacheline[128];
};
int main(void)
{
    printf("sizeof(data) = %ld (1 byte + 127\n"
           "bytes padding + 128-byte array)\n",
           sizeof(struct data));
    printf("alignment of sse_t is %ld\n",
           alignof(struct sse_t));
}
```

Результати виконання:

```
sizeof(data) = 256 (1 byte + 127
bytes padding + 128-byte array)
alignment of sse_t is 16
```

Ще один приклад:

```
#include <stdio.h>
#include <stddef.h>
#include <stdalign.h>
int main(void)
{
    printf("Alignment of char = %ld\n",
           alignof(char));
    printf("alignof(float[10]) = %ld\n",
           alignof(float[10]));
    printf("alignof(struct{char c; int n;}) = %zu\n",
            alignof(struct {char c; int n;}));
}
```

Результати виконання:

```
Alignment of char = 1
alignof(float[10]) = 4
alignof(struct{char c; int n;}) = 4
```

## 8.3. Контрольні запитання

1. Який загальний вигляд структури?
2. Як обчислити розмір структури?
3. Як доступитись до полів структури з змінної типу структура?
4. Як доступитись до полів структури з вказівником на змінну типу структура?
5. Який загальний вигляд об’єднання?
6. Яка принципова різниця між структурою та об’єднанням?
7. Що таке вирівнювання даних?

## 8.4. Лабораторне завдання

1. Навчитися використовувати структури та об’єднання у мові С.
2. Одержати індивідуальне завдання.
3. Побудувати блок-схеми алгоритмів відповідно до завдання.
4. Скласти програми на алгоритмічній мові С згідно завдання.
5. Відлагодити програми, виконати обчислення, проаналізувати отримані результати.

## 8.5. Зміст звіту

1. Титульний аркуш.
2. Мета роботи.
3. Індивідуальне завдання.
4. Блок-схема алгоритмів у відповідності до завдання.
5. Тексти програм у відповідності до завдання.
6. Результати обчислень.
7. Аналіз результатів, висновки.

### 8.6. Індивідуальні завдання

Завдання 1

Скласти програму що дає змогу з використанням структур та об'єднань реалізувати розв’язок поставленої задачі. Всі вхідні дані беруться з текстового файлу (створити не менше десяти відповідних записів у файлі). Ввід-вивд даних та виконання інших окремих логічних дій необхідно реалізувати в окремих функціях. У головній функції необхідно виконувати лише їх виклик. Використання глобальних змінних не допускається. Інформація повинна передаватися у функції лише за допомогою параметрів.

1. В бібліографічному каталозі знайти книги по алгоритмічній мові С. В каталог заноситься шифр книги, прізвище автора, назва, рік видання і кількість сторінок.
2. При обробці інформації про стан здоров’я студентів групи знайти найвищого і найважчого. В медичну картку входить інформація: прізвище, рік народження, ріст, вага.
3. В списку студентів групи обчислити кількість студентів з іменами, що користувач вводить з клавіатури.
4. Знайти найхолодніший і найтепліший день квітня. В метеорологічній інформації міститься: день місяця, температура, % вологості, опади (дощ, сніг тощо).
5. В файлі записана інформація про студентів груп бакалаврату, яка складається з прізвища, імені, статі і віку. Вивести на друк шифр групи в якій найбільший % дівчат.
6. В файлі записано результати екзаменаційного контролю студентів групи. Скласти програму, яка по кількості набраних балів буде визначати оцінку кожного студента. Рахувати що: 100-88 балів це 5; 87-71 балів – 4; 70-51 – 3; менше 50 – 2.
7. В файлі записано прізвище студента і його оцінки за останню сесію. Вивести на друк прізвище і середній бал студентів, як мають середній бал вищий за середній бал групи.
8. Передбачити запис нових абонентів у телефонний довідник сортуючи їх по номеру.
9. У відомості студентів групи записано: Прізвище і ініціали, адреса і телефон. В разі відсутності телефону в графі, що відповідає цій інформації стоять пробіли. Скласти програму яка буде друкувати номер телефону потрібного студента, а в разі відсутності телефону його адресу.
10. По результатах метеорологічних досліджень: дата, середньодобова температура, опади (дощ, сніг тощо), визначити найтепліший день місяця.
11. Передбачити запис нових абонентів у телефонний довідник, якщо вже

записані абоненти розміщені там по алфавіту.

12. В каталозі записано назви книг в алфавітному порядку. В інформацію про книгу входять. Прізвище автора, назва книги, рік видання. Скласти програму яка дозволить доповнювати каталог новими назвами і буде розміщати їх а алфавітному порядку враховуючи і вже існуючі назви.
13. У вхідному файлі записано відомість на стипендію студентів групи, де входить прізвище, ім’я, по-батькові і розмір стипендії. Скласти програму що виведе інформацію про стипендію будь-якого заданого студента.
14. Визначити статистику оцінок, одержаних студентами групи (кількість п’ятірок, четвірок, трійок, двійок) під час сесії.
15. У вхідному файлі записана відомість, де входить прізвище, ініціали, кількість набраних балів. Вивести на друк розсортовану відомість по прізвіщу і по набраних балах та середній бал групи.
16. У вхідному файлі записано телефонний довідник скласти програму, яка по заданому прізвищу буде виводити номер телефону абонента.

Завдання 2

Розробити програму яку забезпечує опрацювання структур даних. Необхідно забезпечити опрацювання 3-5 полів елементів з використанням різних простих типів даних (стрічки, символи, числа). Забезпечити виконання таких операцій:

- додавання нового елементу;
- пошук елементу за значенням полів;
- послідовний перегляд елементів;
- модифікація значень полів елемену;
- видалення елементу;
- сортування за значеннями полів.

Результати всіх операцій повинні зберігатись у файлі (створити не менше десяти відповідних записів у файлі). Елемент:

1. Книга (назва, автор, видавництво, рік, кількість сторінок).
2. Студент (ПІБ, інститут, група, курс, рейтинговий бал).
3. Автомобіль (виробник, модель, ціна, потужність, рік випуску).
4. Мікросхема (марка, розрядність, обсяг, ціна).
5. Фрукт (назва, країна-постачальник, ціна).
6. Музика (назва, виконавець, альбом, жанр, рік).
7. Замовлення в бюро перекладів (клієнт, мова, к-сть. сторінок, перекладач, дата, вартість).
8. Фільм (назва, рік, рейтинг, режисер, жанр, бюджет).
9. Текстові редактори: (назва, виробник, ліцензія, рейтинг, вартість).
10. Відомості про сім'ю студента (ПІБ, інститут, професія батька, професія

матері, кількість братів і сестер).

11. Міський транспорт (вид, номер, початкова зупинка, кінцева зупинка, кількість зупинок, час в дорозі).
12. Насіння (тип, назва рослини, вага насіння, ціна).
13. Побутова техніка (тип, назва, фірма виробника, ціна).
14. Країна (назва, кількість населення, площа).
15. Одяг (тип, назва, розмір, ціна).
16. Фарба (тип, колір, об’єм фасування, ціна).

# СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ


[1] Кормен Т. "Алгоритмы: вводный курс.": Пер. с англ.– М.: ООО "И.Д. Вильямс", 2014.– 208 с.: ил.– Парал. тит. англ. ISBN 978-5-8459-1868-0.

[2] Вирт Н. "Алгоритмы и структуры даннях.": Пер с англ. Ткачев Ф. В.,– М.: ДМК Пресс, 2010.– 272 с.: ил.– ISBN 978-5-94074-584-6.

[3] ГОСТ 19.701-90 / ISO 5807-85.– Єдина система програмної документації. Схеми алгоритмів, програм, даних та систем. Умовні позначення та правила виконання.

[4] Демидович Б., Марон И. "Основы вычислительной математики.".– М.: Наука, 1966, 664 с.

[5] IEEE SA – 1541-2002.– IEEE Standard for Prefixes for Binary Multiples.

[6] Прата С. "Язык программирования С. Лекции и упражнения, 6-е издание.": Пер. с англ.– М.: ООО "Издательский дом Вильямс", 2015.– 928с.: ил.– Парал. тит. англ.– ISBN 978-5-8459-1950-2.

[7] Programming languages – C / N1570 – International standard ISO/IEC 9899:201x – Committee Draft.– April 12, 2011.

[8] Code::Blocks 10.05 Manual Version 1.1 / Code::Blocks The open source, cross-platform IDE [Electronic resource], access mode: <http://www.codeblocks.org/docs/manual_en.pdf>, accessed March 2017.

[9] The IEEE Standard for Floating-Point Arithmetic (IEEE 754-2008). – ISO/IEC/IEEE 60559:2011.

[10] Шеховцов В. "Операційні системи.".– К .: Видавнича група BHV, 2005.– 576 с.: іл. ISBN 966-552-157-8.

[11] The C/C++ Resource Network, 2016 [Electronic resource], access mode: <http://www.cplusplus.com>, accessed March 2017.

[12] Шпак З. Я. "Програмування мовою С, 2-е видання, доповнене". – Львів: Видавництво Львівської політехніки. – 2011.– 436 с.

[13] Інструкції до лабораторних робіт з курсу "Проблемно-орієнтовані мови програмування" для студентів базового напрямку 6.08.04 "Комп'ютерні науки".– Укладачі М. І. Андрійчук, І. І. Чура. – Львів: НУ "ЛП", 2008 р. – 213 с.

НАВЧАЛЬНЕ ВИДАННЯ

**Яворський Назарій Борисович**
**Марікуца Уляна Богданівна**
**Андрійчук Михайло Іванович**
**Фармага Ігор Вірославович**

# ЛАБОРАТОРНИЙ ПРАКТИКУМ З ДИСЦИПЛІНИ АЛГОРИТМІЗАЦІЯ ТА ПРОГРАМУВАННЯ

Навчальний посібник

Редактор _____
Коректор _____
Технічний редактор _____
Комп'ютерне верстання _____
Художник-дизайнер _____

Здано у видавництво _____2018. Підписано до друку _____2018.
Формат 70×100 1/16 . Папір офсетний. Друк офсетний.
Умовн. друк. арк. ___. Обл.-вид. арк. ___.
Наклад _____ прим. Зам. _____

Видавець і виготівник: Видавництво Львівської політехніки
*Свідоцтво суб'єкта видавничої справи ДК № 4459 від 27.12.2012 р.*

*вул. Ф. Колеси, 4, Львів, 79013*
тел. +380 32 2582146, факс +380 32 2582136
vlp.com.ua, ел. пошта: vmr@vlp.com.ua