МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
“ДНІПРОВСЬКА ПОЛІТЕХНІКА”

ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
Кафедра інформаційних технологій та
комп’ютерної інженерії

Гаркуша І.М.

Конспект лекцій
з дисципліни “Програмування”.
Для студентів галузі знань 12 “Інформаційні технології”
спеціальностей
123 “Комп’ютерна інженерія”,
126 “Інформаційні системи та технології”

Дніпро
НТУ “ДП”
2022

УДК 004.4'23+004.42+004.432
Г20

Гаркуша І.М. Конспект лекцій з дисципліни “Програмування”. Для студентів галузі знань 12 “Інформаційні технології” спеціальностей 123 “Комп’ютерна інженерія”, 126 “Інформаційні системи та технології”. 2-ге вид. перероб. і доповн. – Д.: НТУ «ДП», 2022. – 102 с.

В другому виданні конспекту лекцій з дисципліни “Програмування” для студентів спеціальностей 123 “Комп’ютерна інженерія” та 126 “Інформаційні системи та технології” приведений основний матеріал, який викладається у першому семестрі першого курсу навчання за освітнім рівнем “бакалавр”. Додатковий матеріал, який не увійшов до конспекту, додається у теоретико-практичних описах лабораторних робіт.

Основна увага в лекціях приділяється основам програмування мовою С із певним використанням деяких можливостей мови С++.

Протягом лекційного курсу розглядаються приклади побудови простіших консольних програм.

Погоджено рішенням науково-методичних комісій спеціальності 123 Комп’ютерна інженерія (протокол № 6 від 30.06.2022) та спеціальності 126 Інформаційні системи та технології (протокол № 6 від 30.06.2022).

ЗМІСТ

ВСТУП

Метою дисципліни “Програмування” для студентів спеціальностей 123 “Комп’ютерна інженерія” та 126 “Інформаційні системи та технології” є формування компетентностей щодо програмування алгоритмів консольних комп'ютерних програм мовами C/C++.

Всі приклади кодів програм, представлених в конспекті лекцій, а також в завданнях наприкінці конспекту, були скомпільовані за допомогою компілятору GNU GCC. В якості середовища розробки був використаний програмний продукт Code::Blocks. Певні програмні коди можуть бути адаптовані для виконання компілятором Microsoft Visual C++ з деякими невеличкими доробками. Наприклад, якщо студент хоче збирати код у середовищі MS Visual Studio, то на початку більшості приведеного коду, потрібно першим рядком зробити наступний:

#define _CRT_SECURE_NO_WARNINGS

Окрім того, при використанні середовища MS Visual Studio можна також замінювати у наведених прикладах програм певні функції на так звані *safe function*. Наприклад, замість функції *scanf( )* використовувати *scanf_s( )* та т.ін., про які детально описано в документації компанії Microsoft.

Основні дисциплінарні результати навчання, після завершення лекційного курсу “Програмування”:

– використовуючи отриманні знання щодо побудови алгоритмів, представляти їх словесними, формульно-словесними способами та за допомогою схем (блок-схем) при розв’язанні певних обчислювальних задач;

– мати базові знання щодо архітектури обчислювальної техніки, історії її розвитку, систем числення, одиниць вимірювання та представлення даних в пам’яті комп’ютера;

– вміти розробляти простіші консольні програми на базі отриманих знань щодо будування алгоритмів та навичок програмування на мові С;

– реалізовувати обчислення при розробці консольних програм в операційному середовищі MS Windows, роблячи обґрунтований вибір певних структур даних та алгоритмів обробки;

– вміти запрограмувати алгоритми, що реалізовані через консольні програми в операційному середовищі MS Windows, використовуючи мову програмування С та елементи мови програмування C++;

– отримати навички програмування динамічних структур даних, користувацьких типів, а також вміти програмувати операції файлового введення/виведення.

Лекція 1. Введення до алгоритмізації та програмування

Введення в архітектуру комп'ютера

Відповідно до історії розвитку обчислювальної техніки, першою людиною, яка створила лічильну машину, був французький вчений ***Блез Паскаль*** (1623-1662), на честь якого названа одна з мов програмування. Паскаль сконструював цю машину в 1642 році для свого батька (збирача податків) у віці 19 років! Вона була механічна: з шестернями і ручним приводом і могла виконувати тільки операції додавання і віднімання.

Тридцять років по тому великий німецький математик ***Готфрід Вільгельм Лейбніц*** (1646-1716) побудував іншу механічну машину, яка виконувала крім операцій додавання і віднімання також операції множення та ділення.

Через 150 років професор математики Кембриджського університету ***Чарльз Беббідж*** (1792-1871) сконструював механічну різницеву машину, яка виконувала операції додавання, віднімання та підрахунку таблиць чисел для морської навігації. В його машині був закладений тільки один алгоритм – метод кінцевих різниць з використанням поліномів. Після деякого часу, в 1834 році Беббідж розробив механічну аналітичну машину. Ідеї Беббіджа випередили його епоху, і навіть сьогодні більшість сучасних комп'ютерів за будовою подібні його аналітичній машині. Багато хто називає Беббіджа дідусем сучасного цифрового комп'ютера.

У аналітичній машині Чарльза Беббіджа було 4 компоненти: пристрій накопичення (пам'ять), обчислювальний пристрій, пристрій введення (для зчитування перфокарт), пристрій виведення (перфоратор та пристрій для друку). Пам'ять складалася з 1000 слів по 50 десяткових розрядів, кожне з яких містило змінні та результати. Обчислювальний пристрій приймав операнди з пам'яті, потім виконував операції додавання, віднімання, множення або ділення та повертав отриманий результат назад в пам'ять. Оскільки ця аналітична машина програмувалася на асемблері, то їй було необхідно програмне забезпечення. Для його створення Беббідж найняв молоду жінку – ***Аду Августу Ловлейс***, дочку знаменитого британського поета Байрона. Ада була першим у світі програмістом. На її честь названа мова програмування Ада.

З кінця 1930-х років розробки аналітичних машин продовжилися. При розробці почали використовуватися вже електромагнітні реле.

У 1943 році, за участю знаменитого британського математика ***Алана Тьюрінга*** в Великобританії був створений перший електронний цифровий комп'ютер ***COLOSSUS***.

У 1952 році геніальний американський вчений ***Джон фон Нейман*** в Інституті спеціальних досліджень в Прінстоні сконструював машину ***IAS*** (Immediate Address Storage – пам'ять з прямою адресацією). Саме він прийшов

до думки, що програма повинна бути представлена в пам'яті комп'ютера в цифровій формі, разом з даними і для роботи машини доцільніше використовувати не десяткову, а двійкову (бінарну) арифметику.

Основний проект, який фон Нейман описав в своїх дослідженнях, відомий як фон-неймaнська обчислювальна машина (рис. 1.1). Він був використаний в першій машині з програмною пам'яттю ***EDSAC***. Концепція, задум, закладені в проектах IAS та EDSAC зробили дуже великий вплив на подальший розвиток комп'ютерної техніки. Описана ним концепція лежить в основі більшості сучасних цифрових комп'ютерів.

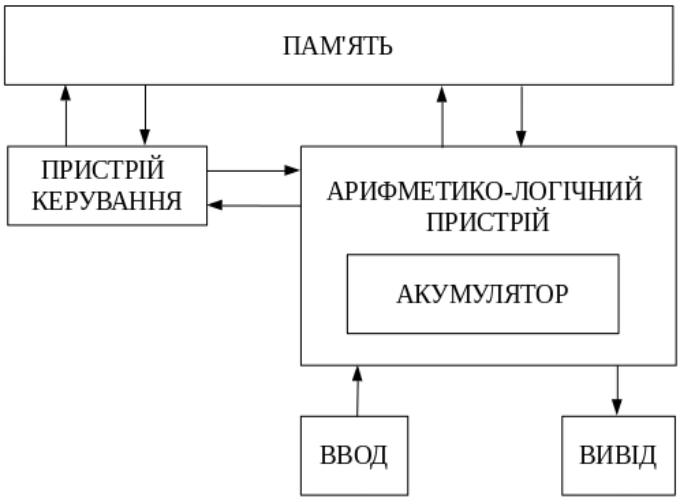


Рис. 1.1. Схематичне уявлення
обчислювальної машини Джона фон Неймана

Будь-яка сучасна комп'ютерна система складається з п'яти основних компонентів: процесора, пам'яті, пристрою управління, пристроїв введення, пристроїв виведення (рис. 1.2). Всі компоненти взаємодіють один з одним за допомогою шин (безлічі об'єднаних ліній зв'язку). Найбільш важлива з них – системна шина, за допомогою якої центральний процесор (ЦП, CPU – Central Processor Unit) взаємодіє з системною пам'яттю та з найважливішими контролерами. Різні контролери можуть виконувати різні дії, наприклад, такі, як введення/виведення (Input/Output, I/O), контроль переривань, контроль доступу до будь-яких зовнішніх пристроїв (USB, накопичувачів на жорстких дисках та ін.) і т.п.

Основне завдання CPU – виконання інструкцій (машинних команд), які перебувають в пам'яті. Машинна команда представляє собою деякий код (певне число), що позначає певну дію CPU. Ці інструкції різні та розрізняються за своїм обсягом, який займають у пам'яті – від одного до декількох байт.

Байт (byte) – одиниця вимірювання обсягу цифрової інформації. Він є найменшою адресною одиницею пам'яті в багатьох архітектурах комп'ютерів. Зазвичай складається з восьми біт. Кожний біт представлений значеннями: 1 або 0.

Сучасні CPU всередині себе містять також свою пам'ять, яка представлена двома видами: кеш-пам'ять (Cache, рівня 1 (L1 Cache) і рівня 2 (L2 Cache)) та регістрова пам'ять. Кеш-пам'ять є копією деякої невеликої області основної пам'яті, до якої CPU найбільш часто звертається.

Регістрова пам'ять представлена регістрами – спеціальними пристроями, що зберігають деяку інформацію. Розрядність значення, що зберігається в регістрі, визначає розрядність регістру.

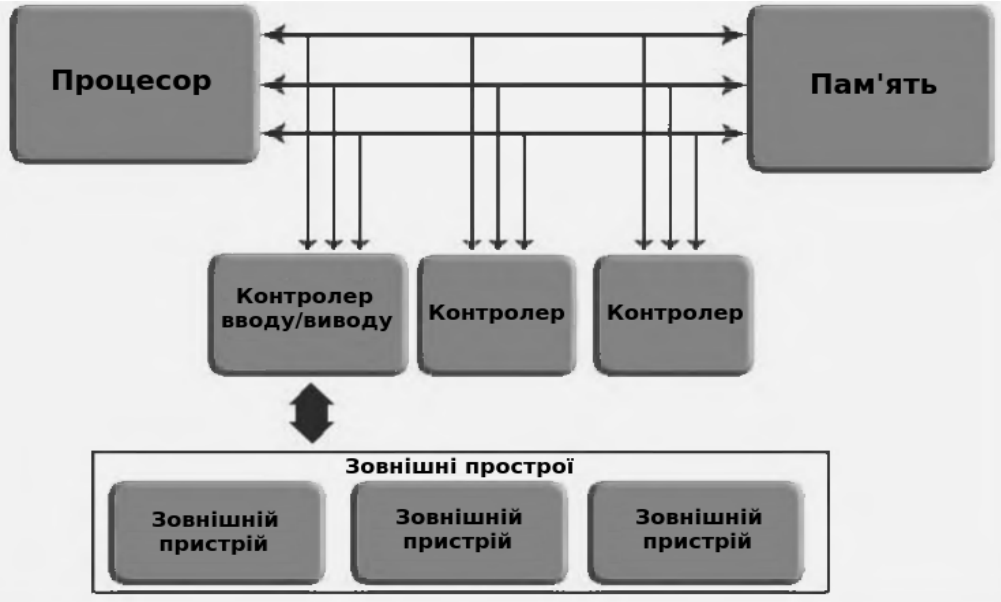


Рис. 1.2. Спрощена схема пристрою сучасного комп'ютера

Серед видатних українських вчених та конструкторів електронно-обчислювальних машин, слід відзначити ***Віктора Михайловича Глушкова*** (1923 – 1982). Він є українським піонером комп'ютерної техніки, автором фундаментальних праць у галузі кібернетики, математики та обчислювальної техніки. Створив велику школу з напряму дослідження та створення обчислювальних систем – засновник інституту кібернетики ім. В.М. Глушкова Національної академії наук України. Під його керівництвом у 1960 році було сконструйовано напівпровідникову ЕОМ “Дніпро”. Слід також відзначити видатного вченого, академіка АН УРСР та СРСР, творця першого в континентальній Європі комп'ютера, ***Сергія Олексійовича Лебедєва*** (1902 – 1974). У 1950 році під його керівництвом у місці Київ була розроблена та створена перша в просторах СРСР та Європі мала електронна обчислювальна машина – МЕОМ. Пізніше у Москві під його керівництвом створено ЕОМ “БЕСМ-1” (велика електронна обчислювальна машина).

Введення в алгоритми, їх класифікація та подання

Перше визначення. Алгоритмом називається система формальних правил, яка чітко та однозначно визначає процес вирішення поставленого завдання у вигляді кінцевої послідовності дій або операцій.

Друге визначення. Алгоритм – це є завершений набір чітких інструкцій по перетворенню інформації (команд), виконання яких призводить до досягнення поставленої мети. Кожна інструкція алгоритму містить точний опис деякої елементарної дії по перетворенню інформації, а також (в явному або неявному вигляді) вказівку на інструкцію, яку необхідно виконувати наступної.

Перша згадка терміну “алгоритм” зустрічається в працях середньоазійського вченого ***Мухаммеда бен Муси аль-Хорезмі*** в IX столітті. Термін використовувався спочатку для визначення правил обчислень в десятковій позиційної системі числення.

Алгоритми повинні мати наступні властивості:

1. Дискретність – процес вирішення розбивається на кроки. Кожен крок – це одна дія або виклик підлеглого алгоритму.
2. Результативність – припускає доступність результату рішення задачі для користувача (друк результату на екран або у файл).
3. Визначеність або детермінованість – запис алгоритму повинен бути чітко визначений. Не допускається неоднозначність тлумачення запису алгоритму.
4. Кінцевість (фінітність) – алгоритм повинен призводити до вирішення завдання обов'язково за кінцевий час.
5. Масовість – вірний результат за алгоритмом повинен бути отриманий на різної безлічі вхідних даних, які припустимі в конкретному завданні.
6. Ефективність – дозволяє вирішити задачу за прийнятний для розробника час при найменших обчислювальних витратах.

Залежно від мети, початкових умов завдання, шляхів його вирішення, визначення дій розробника, алгоритми поділяються на:

1. Керуючи (механічні) – задають певні дії, позначаючи їх в єдиній послідовності, що забезпечує однозначний результат. Наприклад, алгоритми роботи машин, верстатів, двигунів і т.п. Як правило дані для таких алгоритмів надходять від зовнішніх процесів, якими вони керують.
2. Обчислювальні – обробляють порівняно прості види даних, наприклад, числа, вектора, матриці.
3. Інформаційні алгоритми – це колекція простих процедур, що обробляють великі обсяги інформації. Наприклад, процедури виконують пошук необхідної числової або символьної інформації, що відповідає визначеним

ознакам. Причому ефективність роботи таких алгоритмів залежить від організації даних.

Як правило серед обчислювальних та інформаційних алгоритмів можна виділити також ймовірнісні (стохастичні) та евристичні алгоритми.

Ймовірнісні – пропонують програму рішення задачі декількома шляхами або способами, що приводять до досягнення результату.

Евристичні – досягнення кінцевого результату однозначно не визначено, тому що не визначена вся послідовність дій. У таких алгоритмах використовуються універсальні логічні процедури та способи прийняття рішень, засновані на аналогії, асоціаціях і минулому досвіді вирішення схожих завдань. При реалізації евристичних алгоритмів велику роль відіграє інтуїція розробника.

У програмуванні алгоритми поділяються на три типи:

1. Лінійні – набір команд (вказівок), які виконуються послідовно один за одним.
2. Розгалужені – містять хоча б одну перевірку умови, в результаті якої забезпечується перехід на один з можливих варіантів вирішення.
3. Циклічні – передбачають багаторазове повторення однієї і тієї ж дії або операцій над новими вхідними даними.

Способи подання алгоритмів:

1. Словесний – зміст етапів обчислень задається на природній мові в довільній формі з необхідною деталізацією.
2. Формульно-словесний – завдання інструкцій з використанням математичних символів виразів в поєднанні зі словесними поясненнями.
3. Схемний – кожен етап процесу обробки даних представляється у вигляді геометричних фігур (символів), які мають певну конфігурацію в залежності від характеру виконуваних операцій.
4. Псевдокод – сукупність операторів мови програмування та природної мови.
5. Структурні діаграми – використовуються в якості структурних блок-схем для показу міжмодульних зв'язків, а також для відображення структур та систем обробки даних.
6. Мови програмування – фіксовані системи позначень для опису структур даних та алгоритмів, призначені для виконання обчислювальними машинами.

Приклад словесного способу. Завдання 1. Потрібно написати алгоритм обміну значеннями змінних A та B, в разі, якщо A > B.

Крок 1. Отримати значення A.

Крок 2. Отримати значення B.

Крок 3. Порівнюємо: якщо A > B, тоді виконуємо крок 4, інакше – переходимо до кроку 7.

Крок 4. Зберігаємо значення B в тимчасовій змінній C.

Крок 5. Надаємо змінної B значення змінної A.

Крок 6. Надаємо змінної A значення змінної C.

Крок 7. Виводимо значення змінних A та B на екран.

Крок 8. Кінець алгоритму.

Приклад формульно-словесного способу. Завдання 2. Обчислити площу трикутника за трьома сторонами a, b, c.

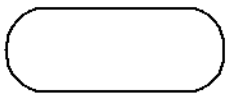
Крок 1. Обчислити напівпериметр трикутника $p = (a+b+c)/2$.

Крок 2. Обчислити площу трикутника $S=\sqrt{p(p-a)(p-b)(p-c)}$.

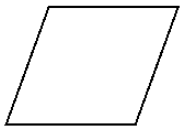
Крок 3. Вивести на екран результат S та припинити обчислення.

При описі алгоритму схемами використовують зображення безлічі стандартних символів в графічному поданні, детально описаних в стандарті ДСТУ ISO 5807:2016 “Обробляння інформації. Символи та угоди щодо документації стосовно даних, програм та системних блок-схем, схем мережевих програм та схем системних ресурсів” (ISO 5807:1985, IDT).

Основними символами вважаються наступні:

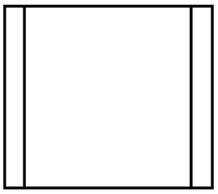


Термінатор (Terminator) – відображає вихід в зовнішнє середовище та вхід із зовнішнього середовища (початок або кінець схеми програми, зовнішнє використання та джерело або пункт призначення даних).

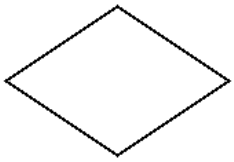


Дані (Data) – відображає дані, носій даних не визначено.

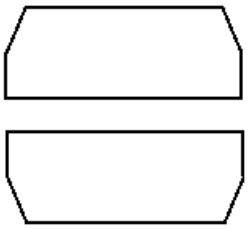
Процес (Process) – відображає функцію обробки даних будь-якого виду (виконання певної операції або групи операцій, що приводить до зміни значення, форми або розміщення інформації або до визначення, за яким з декількох напрямків потоку слід рухатися).



Попередньо визначений процес (Predefined process) – процес, що складається з однієї або декількох операцій або кроків програми, визначених у іншому місці (в підпрограмі, модулі).



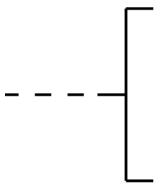
Рішення (Decision) – відображає рішення або функцію перемикача типу, що має один вхід та ряд альтернативних виходів, один і тільки один з яких може бути активізований після обчислення умов, визначених всередині цього символу. Відповідні результати обчислення можуть бути записані по сусідству з лініями, що відображають ці шляхи.



Межа циклу (Loop limit) – складається з двох частин, що відображають 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, перевіряє умову.

Підготовка (Preparation) – відображає модифікацію команди або групи команд з метою впливу на деяку подальшу функцію (установка перемикача, модифікація індексного регістру або ініціалізація програми).

З'єднувач (Connector) – відображає вихід в частину схеми та вхід з іншої частини цієї схеми й використовується для обриву лінії та продовження її в іншому місці. Відповідні символи-з'єднувачі повинні містити одне і теж унікальне позначення.



Коментар (Comment) – використовують для додавання описових коментарів або пояснювальних записів з метою пояснення або приміток. Пунктирні лінії в символі пов'язані з відповідним символом або можуть обводити групу символів. Текст коментарів або приміток повинен бути поміщений біля обмежуючої фігури.

Приклад подання завдання 1 за допомогою схеми алгоритму показано на рис. 1.3.

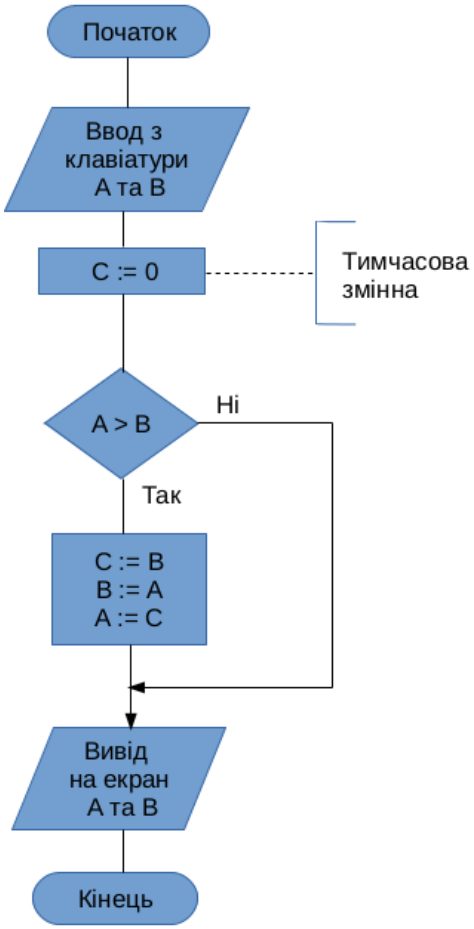


Рис. 1.3. Приклад подання алгоритму за допомогою схеми

Введення в програмування та мови

Мова програмування – це певна знакова система для планування поведінки комп'ютера.

Комп'ютерна програма являє собою опис тієї послідовності дій, яку повинен виконати комп'ютер. Цей опис, з точки зору машини, виглядає як послідовність машинних команд, реалізація яких закладена в електронні схеми процесору комп'ютера.

Сукупність машинних команд становить так звану машинну мову або мову низького рівня.

Для різних апаратних процесорних архітектур існують різні, часто несумісні, машинні команди. Наприклад, машинні команди процесорів архітектури x86 та ARM різні.

Мови програмування *високого рівня* призначені насамперед для зручності та швидкості створення й розвитку програм програмістами. Вони не призначені

для прямого виконання на процесорах комп'ютера. Для виконання на процесорах різних архітектур їх необхідно переводити до форми подання машинними командами. Кажуть, що перед виконанням такі програми повинні бути піддані трансляції.

Транслятором називають програму, яка в якості вхідних даних сприймає програми деякою вхідною мовою, а на виході формує еквівалентні за своєю функціональністю програми, але вже на іншій, так званій об'єктній мові (не має відношення до так званих об'єктних мов програмування високого рівня). Наприклад, *Асемблером* називають транслятор, у якого об'єктною мовою є деякий різновид машинної мови будь-якого апаратного комп'ютера, а вихідною мовою – символічне уявлення машинної мови. Вхідну мову зазвичай називають мовою асемблера. Найчастіше кожна команда мовою оригіналу перекладається в одну команду на об'єктній мові.

Відомі дві основні різновиди трансляторів: компілятори та інтерпретатори.

Компілятори спочатку повністю переводять весь текст програми з мови високого рівня в мову машинних команд для подальшого запуску отриманої програми. При цьому в подальшому користувачеві не потрібен вхідний код і він може багаторазово запускати отриману програму на комп'ютері.

До найбільш відомих мов програмування з компіляторами відносять, наприклад, C, C ++, Objective-C, Pascal, Object Pascal, Fortran та ін.

Першою високорівневою мовою програмування з компілятором, яка отримала широке застосування, був Fortran (IBM, 1957 рік). Мова була створена в період з 1954 по 1957 рік групою програмістів під керівництвом ***Джона Бекуса*** в корпорації IBM. Вона призначалася для наукових та технічних розрахунків.

Мова C була розроблена в 1972 році ***Денісом Рітчі*** та ***Кеном Томпсоном*** (AT&T Bell Telephone Laboratories).

Інтерпретатори обробляють текст коду програми на мові високого рівня і виконують його в міру обробки (читання). Переклад програми на машинну мову не запам'ятовується. Тому для багаторазового повторення виконання програми доводиться знову запускати її код через інтерпретатор.

До найбільш відомих мов програмування з інтерпретатором відносять, наприклад, Python, Ruby, Matlab та ін.

Є мови програмування, для яких були розроблені як інтерпретатори, так і компілятори. Наприклад, мова програмування BASIC – свого часу був розроблений інтерпретатор Altair BASIC (1975) для комп'ютера Altair 8800, а також серед багатьох реалізацій можна пригадати компілятор FreeBASIC (2004).

Зауважимо, що є мови, назви яких є акронімами, або скороченнями, а деякі отримали свою назву на честь певних світових вчених, або за назвою певної географічної місцевості. Прикладом мови програмування з назвою, що є акронімом, є все той же BASIC – це акронім від слів Beginner's All-purpose Symbolic Instruction Code (універсальний код символьних інструкцій для початківців). Назву цієї мови спрощено також називають як початкова, елементарна. Ще прикладом є мови Fortran – це скорочення від FORmula TRANslator (перекладач формул), а також ALGOL (ALGOrithmic Language).

Найвідоміші приклади мов програмування з назвами на честь вчених або географічних міст: Pascal (на честь Блеза Паскаля), Ada (на честь Августи Ади Кінг), Karel (на честь Карела Чапека, який придумав слово “робот”), Haskell (на честь американського математика Гаскелла Каррі), Kotlin (назва походить від Kotlin Island неподалік від Санкт-Петербургу).

Існує ряд сучасних мов програмування, які використовують компілятори для перекладу тексту коду мови високого рівня в проміжний, але не в машинний, а так званий *проміжний байт-код*, який в подальшому буде інтерпретований певною програмою, яку часто називають віртуальною машиною. Такі віртуальні машини для виконання проміжних байт-кодів збирають під певні апаратні платформи. Найбільш відомими представниками цього сімейства мов, що використовують свої власні віртуальні машини виконання проміжних байт-кодів, є мови Java (використовує Java Virtual Machine) та C# (використовує Common Language Runtime).

Оскільки багато мов програмування розвиваються, то у багатьох існують версії, а також діалекти. наприклад:

– мова C – ANSI C, C99, C11 та ін.;

– мова C++ – C++ 98, C++03, C++11 та ін.;

– мова Fortran – FORTRAN 66, FORTRAN 77, Fortran 2003, Fortran 2008;

– мова Python – Python 2, Python 3.

Та т.ін.

Окрему категорію займають сценарні (скриптові) мови програмування, які фактично належать до інтерпретаторів, але ставлять перед собою за мету дати можливість створювати програми, що використовують більш високорівневі готові програмні компоненти та конструкції для вирішення завдань в контекстах певних середовищ. Наприклад, в середовищі Web-браузера або в операційній системі (ОС). Прикладами можуть служити ECMAScript (приклади діалектів: JavaScript, ActionScript), VBScript, PHP, Perl, Bash, PowerShell. Багато фахівців також схильні відносити до цієї категорії мови Python та Ruby.

Розвиток світових найвідоміших мов програмування представлений окремою сторінкою у проекті Wikipedia:

https://en.wikipedia.org/wiki/Timeline_of_programming_languages

Головною ідеєю будь-якого складного поняття, в тому числі і мови програмування, є парадигма. Стосовно до мов програмування розрізняють наступні парадигми.

1. Імперативне (процедурне) програмування.
2. Функціональне програмування.
3. Логічне програмування.
4. Об'єктне програмування.
5. Об'єктно-орієнтоване програмування.

Імперативні мови програмування орієнтовані на оператори. Обчислення в них представляються як послідовність дій, вироблених операторами.

Функціональні мови програмування задають обчислення як виклики функцій.

Логічні мови програмування описують обчислення за допомогою формальної логіки.

В об'єктних та об'єктно-орієнтованих мовах програмування обчислення реалізуються сукупністю об'єктів.

Розглянемо більш докладно парадигму імперативних мов програмування. До імперативних мов відносять: Fortran, ALGOL, COBOL, PL/1, Pascal, C та ін. Вони орієнтовані на комп'ютери з архітектурою фон Неймана. Основні поняття імперативних мов програмування тісно пов'язані з компонентами комп'ютера:

– змінні різних типів (моделюють осередки пам'яті);

– оператори присвоювання (моделюють пересилання даних);

– повторення дій в формі ітерації (моделюють зберігання інформації в суміжних комірках пам'яті).

При реалізації оператора присвоювання відбувається наступне: операнди виразу передаються з пам'яті в процесор, а результат обчислення виразу заноситься в комірку пам'яті, ім'я якої вказано в лівій частині оператора. Оскільки операнди зберігаються в сусідніх комірках пам'яті, то ітерації на комп'ютері фон Неймана здійснюються дуже швидко.

Обчислення ґрунтуються на понятті стан комп'ютера (контекст процесу виконання). Це безліч всіх значень всіх осередків його пам'яті. Програма складається з послідовності операторів, виконання кожного з яких тягне за собою зміну значення в одній або декількох осередках пам'яті, тобто перехід комп'ютера в новий стан. У загальному випадку синтаксис імперативною програми має вигляд:

```
оператор1;
оператор2;
. . .
```

Найчастіше оператори виконуються в порядку їх слідування в програмі, один за одним, і призводять до послідовної зміни станів комп'ютера. Кінцевий стан забезпечує необхідний результат.

Лекція 2. Системи числення. Одиниці вимірювання інформації та представлення даних в пам'яті комп'ютера

Числова інформація в комп'ютерах характеризується:

– системою числення (двійкова, десяткова і ін.);

– видом числа (дійсні числа, комплексні, масиви);

– типом числа (змішане, ціле, дробове);

– формою подання числа (місце коми) – з природною (змінною), фіксованої, рухомою (плаваючою) комою;

– розрядною сіткою та форматом числа;

– діапазоном та точністю представлення чисел;

– способом кодування від'ємних чисел – прямим, зворотним та додатковим кодами;

– алгоритмами виконання арифметичних операцій.

Системою числення називається сукупність чисел і правил для запису чисел. Запис числа в деякій системі числення називається його *кодом*.

У будь-якій системі числення для подання чисел вибираються деякі символи (слова або знаки), які називають *базисними числами*, а всі інші числа отримують в результаті будь-яких операцій з базисних чисел даної системи числення. Наприклад, для десяткової системи числення базисними є такі арабські цифри: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

Системи числення, в яких будь-яке число виходить шляхом додавання або віднімання базисних чисел, називаються *адитивними*.

Всі системи числення діляться на *позиційні* та *непозиційні*. Для запису чисел в позиційній системі числення використовують певну кількість графічних знаків (цифр і букв), які відрізняються один від одного. Число таких знаків називають *основою позиційної системи числення*. У комп'ютерах використовують позиційні системи з різною основою.

Основа	*Система числення*	*Знаки*
2	Двійкова	0, 1
8	Вісімкова	0, 1, 2, 3, 4, 5, 6, 7
10	Десяткова	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
16	Шістнадцяткова	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

У непозиційних системах числення значення кожної цифри не залежить від її позиції. Найвідомішою непозиційною системою є римська, в якій

використовуються сім знаків – I, V, X, L, C, D, M, що відповідають таким значенням:

I	V	X	L	C	D	M
1	5	10	50	100	500	1000

Наприклад: III – 3, LIX – 59, DLV – 555.

Число в позиційній системі можна представити поліномом:

$$A_q = a_k \cdot q^k + a_{k-1} \cdot q^{k-1} + ... + a_0 \cdot q^0 + a_{-1} \cdot q^{-1} + ... + a_{-m} \cdot q^{-m} = \sum_{i=-m}^{k} a_i \cdot q^i$$

де q – основа системи числення; q^i – вага позиції; $a_i \in \{0,1,\ldots,(q-1)\}$ – цифри в позиціях числа; 0, 1,..., k – номери розрядів цілої частини числа; –1, –2, ..., –m – номери розрядів дробової частини числа.

Позиційні системи з однаковою основою в кожному розряді називають *однорідними*. Оскільки на значення q немає ніяких обмежень, то теоретично можливо безліч позиційних систем числення.

Перевагою двійкової системи є: простота виконання арифметичних операцій; наявність надійних мікроелектронних схем з двома стійкими станами (їх називають тригерами), призначених для зберігання значень двійкового розряду – цифр 0 або 1. Двійкові цифри називають також *бітами*. Слово «біт» походить від англійських слів *binary* (двійковий) та *digit* (цифра): BInary + digiT = BIT.

Розряди двійкового числа характеризуються вагою, яка кратна ступеню двійки – 1, 2, 4, 8, ... (в напрямку від молодших до старших розрядів). Наприклад, для чотирьох розрядного двійкового числа маємо:

$$x = x_3 \cdot 2^3 + x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0 = x_3 \cdot 8 + x_2 \cdot 4 + x_1 \cdot 2 + x_0 \cdot 1$$

Якщо двійкове число x_2=1101, то отримуємо такий десятковий еквівалент: x_{10} = 1*8 + 1*4 + 0*2 + 1*1 = 13.

У двійково-десятковій системі числення кожна десяткова цифра записується чотирма двійковими розрядами (*тетрадами*). Наприклад:

$$A_{10} = 873{,}25 = \begin{array}{ccccccccc} 1000 & 0111 & 0011, & 0010 & 0101 \\ 8 & 7 & 3 & 2 & 5 \end{array}$$

Значення від 0 до 20 в різних системах числення представлені в таблиці 2.1. Кожна трійка двійкових розрядів відповідає одній вісімковій цифрі, кожна четвірка – шістнадцятковій.

Таблиця 2.1

Значення від 0 до 20 в різних системах числення

q = 10	*q = 2*	*q = 8*	*q = 16*
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10
17	10001	21	11
18	10010	22	12
19	10011	23	13
20	10100	24	14

Біти даних групують в набори. Однією зі стандартних одиниць виміру обсягу даних є *байт* – *об'єднання восьми біт*.

Кількість різних комбінацій цифр, яка може бути представлена в одному байті, так само: $2^8 = 256$.

Як правило розрядність сучасної регістрової пам'яті центральних процесорів значно вище і досягає 32, 64, а в регістрах спеціального призначення – 128 та 512 розрядів.

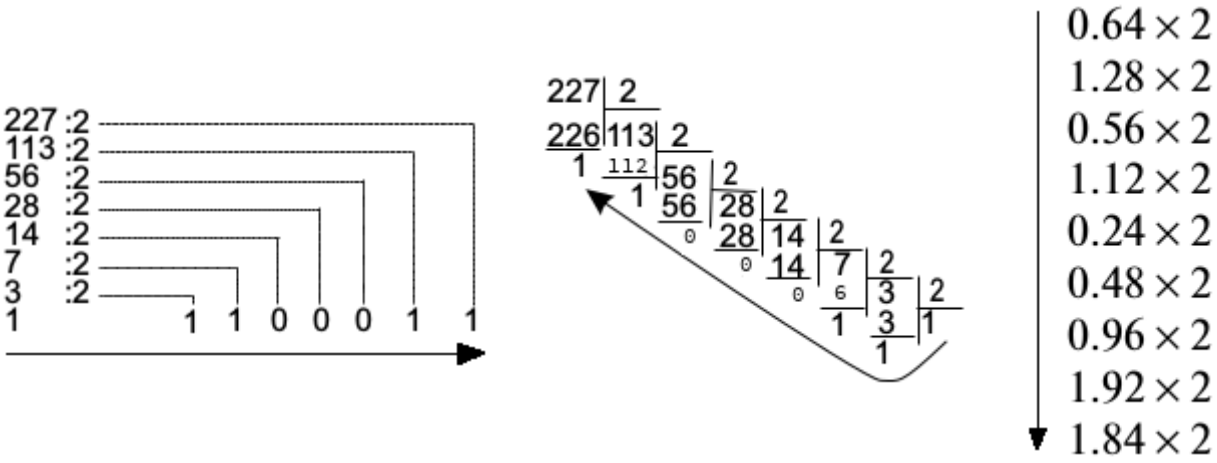
Об'єднання з 2-х байт (16 біт) називають *машинним словом (word)*, з 4-х байт (32 біта) – *подвійним машинним словом (dword, double word)*, з 8-ми байт (64 біта) – *машинним квадрословом (qword, quad word)*.

Переклад числа з десяткової системи числення в двійкову

Цей переклад здійснюється окремо для цілої та дробової частин числа за такими алгоритмами.

1. Ціле десяткове число ділиться без остачі на основу 2, потім на 2 діляться послідовно всі частки від цілочисельного ділення, до тих пір поки частка не стане менше основи. В результат заносяться остання частка і всі залишки від ділення, починаючи з останнього (рис. 2.1, а).

2. Десяткова дріб послідовно множиться на основу 2, причому відразу після кожної операції множення отримана ціла частина записується в результат і в подальшому збільшенні участі не бере. Кількість операцій множення залежить від необхідної точності (рис. 2.1, б).



а) б)

Рис. 2.1. Переклад чисел з десяткової в двійкову систему числення:
а) $227_{10} = 11100011_2$; б) $0{,}64_{10} = 0{,}10100011_2$

Переклад числа з двійкової системи числення в десяткову

Його можна здійснити для цілої і дробової частин числа по одному алгоритму шляхом обчислення суми добутків цифри двійкового числа на вагу її знакомісця. Наприклад:

$$11100011_2 = 1x2^7 + 1x2^6 + 1x2^5 +$$
$$+0x2^4 + 0x2^3 + 0x2^2 + 1x2^1 + 1x2^0 =$$
$$128 + 64 + 32 + 2 + 1 = 227_{10}$$

$$0{,}10100011_2 = 1x2^{-1} + 0x2^{-2} + 1x2^{-3} +$$
$$+0x2^{-4} + 0x2^{-5} + 0x2^{-6} + 1x2^{-7} + 1x2^{-8} =$$
$$0{,}5 + 0{,}125 + 0{,}0078125 + 0{,}00390625 = 0{,}63671875_{10} \approx 0{,}64_{10}$$

Подання в комп'ютері від'ємних чисел

У комп'ютерах восьмибітовий регістр є найменшим і власне він і представляє найменшу двійкове число, яке може зберігатися в пам'яті, тобто Воно повинно бути восьмибітовим й представлено, як правило, типом даних byte (аналог в мові *C* (Сі) – *unsigned char*). При цьому в незаповнені клітинки регістру (в старших розрядах) записуються нулі.

На відміну від десяткової, в двійковій системі числення відсутні спеціальні символи, що позначають знак числа (+ або –), тому для подання двійкових від'ємних чисел використовуються наступні дві форми.

1. Форма значення зі знаком: старший (лівий) розряд помічається як знаковий і містить інформацію лише про знак числа:

0 – число позитивне;

1 – число від'ємне.

Решта розрядів відводяться під абсолютну величину числа:

$5_{10} = 0000\ 0101_2$
$-5_{10} = 1000\ 0101_2$

2. Форма зворотнього додаткового коду, переклад в яку проводиться за наступним алгоритмом:

– інвертувати всі розряди числа, крім знакового розряду;

– додати одиницю до отриманого коду;

– відновити одиницю в знаковому розряді.

$$-5_{10} = 1000\ 0101 \rightarrow 111\ 1010 + 1 \rightarrow 111\ 1011 \rightarrow 1111\ 1011_2$$

Комп'ютери зазвичай будуються таким чином, щоб від'ємні числа були представлені в додатковому коді, оскільки це дає суттєву економію часу при виконанні з ними арифметичних операцій. Основні властивості додаткових кодів:

– додатковий код позитивного числа – саме число;

– перетворення додаткового коду за наведеним алгоритмом перекладу призводить до первинного вигляду числа в знаковій формі.

Подання чисел в форматі з фіксованою точкою

Фіксованою називається така точка, позиція якої фіксується форматом представлення чисел. Це такий спосіб поділу цілої та дробової частин числа, при якому положення точки не змінюється в ході виконання операцій над даним числом. При розміщенні числа з фіксованою точкою в слові для його уявлення використовуються розряди з 0-го по 14-й. Знак числа міститься в розряді 15. Від'ємні числа в форматі з фіксованою точкою представляються, як правило, в додатковому коді (за допомогою операції доповнення до двох).

Діапазон представлення чисел в форматі з фіксованою точкою: для байту – від -128_{10} до $+127_{10}$, для слова – від -32768_{10} до $+32767_{10}$.

Подання чисел в форматі з рухомою точкою

Рухомою називається така точка, позиція якої не фіксується форматом числа. Будь-яке дійсне число X, представлене в системі числення з основою N, можна записати у вигляді:

$$X = \pm mN^{\pm p},$$

де m – мантиса, p – характеристика (порядок) числа.

Якщо $|m| < 1$, то запис числа називається нормалізованим зліва. При нормалізації справа, після коми в мантисі буде не нуль. Наприклад, число $0{,}00076_{10}$ буде нормалізованим справа, якщо представлено у вигляді $0{,}76 \times 10^{-3}$, а $0{,}076 \times 10^{-2}$ не є таким. Інші приклади нормалізованих чисел:

$$372.95 = 0.37295 \times 10^{3},$$
$$25 = 0.025 \times 10^{3} = 0.25 \times 10^{2},$$
$$0.0000015 = 0.15 \times 10^{-5} = 0.015 \times 10^{-4}$$

Як правило в комп'ютерах використовується нормалізація справа.

Детально ознайомитися з прикладом представлення чисел з рухомою точкою можна за посиланнями:

https://en.wikipedia.org/wiki/Single-precision_floating-point_format
https://en.wikipedia.org/wiki/Double-precision_floating-point_format
https://en.wikipedia.org/wiki/Quadruple-precision_floating-point_format

Порядок запису байтів

У персональних комп'ютерах на базі Intel-сумісних процесорів прийнятий little-endian порядок запису байтів даних в пам'яті (від молодшого до старшого). Такий порядок запису байт іноді називають інтеловським або як VAX order.

Також поширений порядок записи байт від старшого до молодшого – big-endian, який використовується в мережевих протоколах TCP/IP, а також на комп'ютерах з процесорами IBM 360/370/390, Motorola 68000, SPARC.

Приклад зберігання значень 4-хбайтового числа у зворотньому порядку:

12_{10} = $0C\ 00\ 00\ 00_{16}$ ($00\ 00\ 00\ 0C_{16}$ – в прямому порядку)

-12_{10} = $F4\ FF\ FF\ FF_{16}$ ($FF\ FF\ FF\ F4_{16}$ – в прямому порядку)

32767_{10} = $FF\ 7F\ 00\ 00_{16}$ ($00\ 00\ 7F\ FF_{16}$ – в прямому порядку)

-32767_{10} = $01\ 80\ FF\ FF_{16}$ ($FF\ FF\ 80\ 01_{16}$ – в прямому порядку)

2147483647_{10} = $FF\ FF\ FF\ 7F_{16}$ ($7F\ FF\ FF\ FF_{16}$ – в прямому порядку)

-2147483648_{10} = $00\ 00\ 00\ 80_{16}$ ($80\ 00\ 00\ 00_{16}$ – в прямому порядку)

Останні два значення – це граничні значення, які можна уявити за допомогою 4-х байт. При додаванні одиниці до передостаннього числа або відніманні одиниці з останнього числа буде досягнуто переповнення розрядної сітки і значення в байтах буде скинуто в передостанньому випадку на -2147483648, а в останньому випадку на 2147483647. Тому при організації програм на комп'ютері дуже важливим є вибір типу даних що зберігають значення. ***Саме тип даних і визначає розрядність пам'яті, яку компілятор відводить для зберігання цих значень.***

Приклад перекладу -12_{10}:

$12_{10} = C_{16} = 1100_2$ → (інверсія біт) 0011_2 → (+1) 0100_2 → 4_{16} → (+біт знаку) $1111\ 0100_2 = F4_{16} = -12_{10}$

Подання символьної інформації в комп'ютері

Образ кожного графічного символу, що відображається на екрані монітора представлений в пам'яті операційних систем (ОС) у вигляді таблиць відповідності: *код – графічний символ*. Такі таблиці називають кодувальними (кодовими) таблицями символів. За замовчуванням в ОС сімейства MS-DOS та

сумісних з нею, використовувалася кодувальна таблиця символів ASCII. Код кожного символу з таблиці займав в пам'яті один байт. Всього можна було уявити 256 різних символів (від 0 до 255 – 2^8). Таблиця ділилася на дві частини. Перша частина з кодами від 0 до 127 містила визначені розробником ОС і стандартні символи латиниці, арабські цифри, неграфічні символи (наприклад, пробіл) і деякі допоміжні графічні символи. Коди від 128 до 255 варіювалися в так званих кодових сторінках для представлення різних іноземних мов, відмінних від англійської. Так, наприклад, кодова сторінка 866 описувала символи кирилиці.

У перших системах ОС MS Windows була використана кодувальна таблиця ANSI (256 символів), в якій символи з кодами від 0 до 127 були практично ті ж, що і в ASCII-таблиці, а кодові сторінки для кодів від 128 до 255 вже відрізнялися від варіанту MS-DOS. Кодові сторінки для кирилиці у варіанті ANSI мають номери 1251 та 1252.

У той же час, на безлічі Unix-подібних ОС було поширене кодування KOI-8 – восьмибітова кодова сторінка, сумісна з ASCII. KOI8-U, KOI8-R – містили літери українського та російського алфавітів. У MS Windows цим кодуванням відповідають кодові сторінки 21866 та 20866.

З подальшим розвитком ОС MS Windows (починаючи з версії Windows NT 3.1 (1993 рік) та Windows 95 (1995 рік)) набуло поширення кодувальна таблиця Unicode, в якій кожен символ представлений двома байтами (65536 символів – 2^{16}). Коди в цьому стандарті розділені на кілька областей. Область з кодами від U+0000 до U+007F містить всі ті ж символи набору ASCII.

Існує також безліч різних форм представлення Unicode. Наприклад, найбільш відомими є UTF-8, UTF-16 та UTF-32.

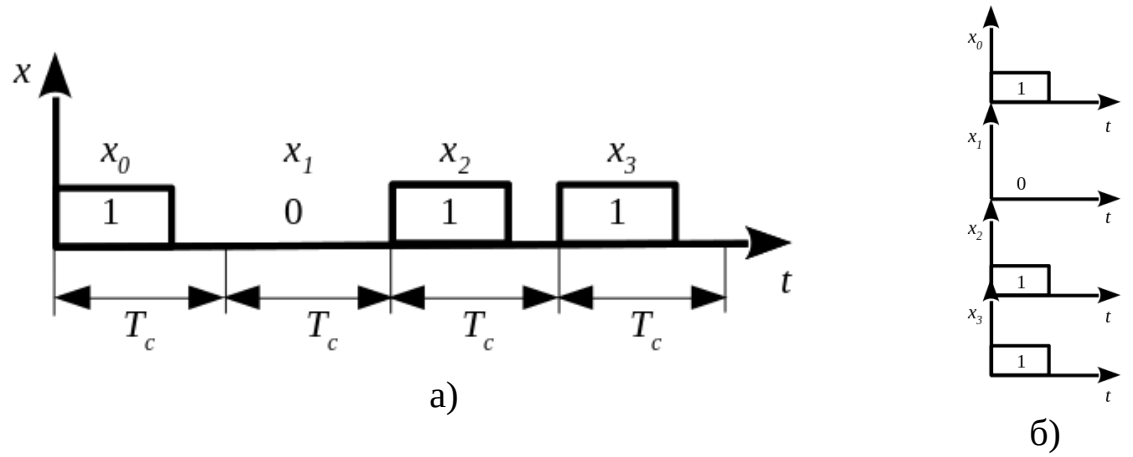
В мовах програмування C/C++ для роботи з символами в однобайтному та двобайтному кодуваннях використовуються різні типи даних і позначення.

Передача інформації та одиниці вимірювання

Цифровий код в комп'ютері передають по лініях зв'язку (по шинам передачі) послідовно в часі (послідовний код) за допомогою одного каналу передачі (рис. 2.1, а) або одночасно (паралельний код) за допомогою багатоканальної передачі (рис. 2.1, б).

На практиці послідовний код використовують при передачі інформації на великі відстані (наприклад, між комп'ютерами), а паралельний код – при передачі інформації на малі відстані (наприклад, передачі всередині комп'ютеру).

При передачах біт в послідовних лініях зв'язку використовують одиницю швидкості біт/с, при паралельній передачі даних – байт/с.



а) б)

Рис. 2.1. Передача інформації (t – вісь часу, x – вісь амплітуди сигналу): а) – послідовним кодом (T_c – деякий часовий проміжок формування імпульсу сигналу); б) – паралельним кодом

Крім біта та байта для вимірювання кількості інформації використовуються також більш крупні одиниці:

1 кілобайт = 1024 байт
1 мегабайт = 1024 Кбайт
1 гігабайт = 1024 Мбайт
1 терабайт = 1024 Гбайт

Існує й інша система позначень:

Вимірювання в байтах					
Десяткова приставка			Двійкова приставка		
Назва	Символ	Ступінь	Назва	Символ	Ступінь
		ДСТУ			МЕК*
байт	B	10^0	байт	B байт	2^0
кілобайт	kB	10^3	кібібайт	KiB Кбайт	2^{10}
мегабайт	MB	10^6	мебібайт	MiB Мбайт	2^{20}
гігабайт	GB	10^9	гібібайт	GiB Гбайт	2^{30}
терабайт	TB	10^{12}	тебібайт	TiB Тбайт	2^{40}
петабайт	PB	10^{15}	пебібайт	PiB Пбайт	2^{50}
ексабайт	EB	10^{18}	эксбібайт	EiB Эбайт	2^{60}
зеттабайт	ZB	10^{21}	зебібайт	ZiB Збайт	2^{70}
йоттабайт	YB	10^{24}	йобібайт	YiB Йбайт	2^{80}

* МЕК – Міжнародна електротехнічна комісія – в березні 1999 року ввела стандарт МЕК 60027-2, в якому описано іменування двійкових чисел. Аналогічний стандарт IEEE 1541-2002 введений в 2008 р.

Лекція 3. Введення в мову програмування С

Коротка історія появи UNIX та мови програмування С

У 1965 році *Bell Telephone Laboratories* (підрозділ AT&T) спільно з *General Electric Company* та *Масачусетським технологічним інститутом* (MIT) почали розробляти нову операційну систему (ОС), яка отримала назву *MULTICS* (MULTiplexed Information and Computing Service). Перед учасниками проекта стояла мета створення багатозадачної ОС поділу часу, здатної забезпечити одночасну роботу кількох сотень користувачів. Від Bell Labs в проекті взяли участь два співробітника – ***Кен Томпсон*** (Ken Tompson, 1943) та ***Денніс Рітчі*** (Dennis Ritchie, 1941-2011). Хоча система *MULTICS* так і не була завершена (в 1969 році Bell Labs вийшла з проекту), вона стала предтечею ОС, що згодом отримала назву *UNIX* – класичної мережевої ОС. Томпсон, Рітчі та ряд інших співробітників продовжили роботу над створенням зручного середовища програмування. Використовуючи ідеї та розробки, що з'явилися в результаті роботи над *MULTICS*, вони створили в 1969 невелику ОС, що включала файлову систему (ФС), підсистему управління процесами та невеликий набір утиліт. Система була написана на асемблері і застосовувалася на комп'ютері PDP-7. Ця ОС отримала назву *UNICS* (***UNiplexed Information and Computing System***), співзвучне *MULTICS* і придумане іншим учасником групи супроводу, ***Брайаном Керніганом*** (Brian Kernighan, 1942) та надалі скорочена до ***UNIX***.

Рис. 3.1. Martin Richards, Brian Kernighan, Dennis Ritchie, Ken Tompson

У 1973 році ядро ОС було переписано на мові високого рівня С для машин PDP-11, – нечуваний до цього крок, що зробив величезний вплив на популярність UNIX. Це означало, що тепер система UNIX може бути перенесена на інші апаратні платформи за лічені місяці, крім того, значна модернізація системи та внесення змін не представляли особливих труднощів.

Багато важливих ідей мови С взяті з мови *BCPL* (Basic Combined Programming Language), автором якої є ***Мартін Річардс*** (Martin Richards, 1940). Вплив BCPL на С був непрямим – через мову програмування В, розробленою

Кеном Томпсоном в 1970 році для першої UNIX, реалізованої для обчислювальної машини PDP-7.

Перший опис мови С було дано його авторами, Б. Керніганом і Д. Рітчі на початку 70-х років. У 1978 році вийшла перша книга авторів «Мова програмування Сі», а в 1983 році Американський інститут національних стандартів – ANSI, заснував комітет зі стандартизації мови. Результатом його роботи став випуск стандарту «ANSI-C» в 1988 році.

Рис. 3.2. Різновиди PDP-11. 16-розрядна міні-ЕОМ компанії DEC

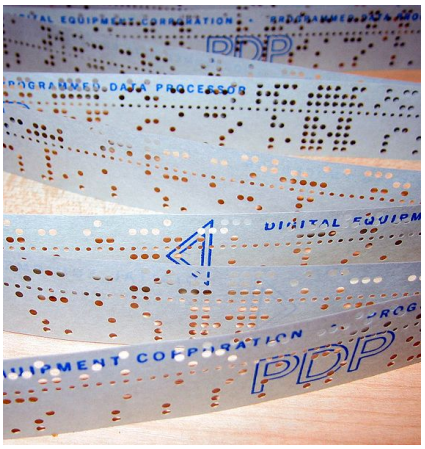


Рис. 3.3. Перфострічка, яка використовувалася у PDP-11

Мова С – універсальна мова програмування, що займає проміжне положення між низькорівневими та високорівневими мовами програмування. Хоча вона пов'язана з ОС UNIX, але вона жорстко не прив'язана до неї. Вона ідеально підходить не тільки як мова системного програмування для розробки компіляторів та ОС, але і як мова прикладного програмування. Саме мова компактна та не має вбудованих механізмів введення-виведення (I/O), управління пам'яттю, прямих операцій над такими об'єктами, як рядки, масиви, списки, множини. Всі ці механізми високого рівня забезпечуються виключно через виклик бібліотечних функцій.

На рис. 3.4 винесені назви стандартів мов програмування C/C++ з зазначенням років їх прийняття. Різні компілятори мов C/C++ за замовчуванням підтримують різні стандарти.

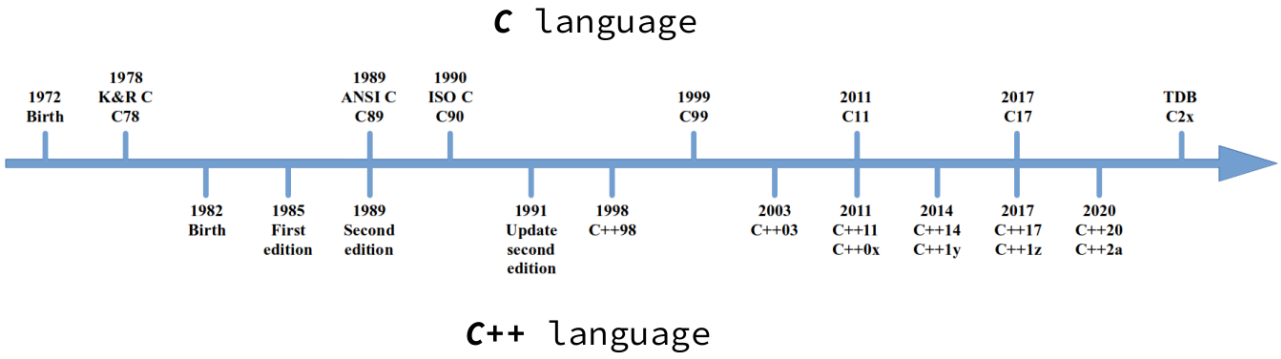


Рис. 3.4. Розвиток мов програмування C/C++ в часі – поява відповідних стандартів

Фази підготовки програми. Основні терміни

Основні фази підготовки програми на мові C/C++ від редагування коду до формування бінарного файлу програми для виконання в ОС, представлені на рис. 3.5.

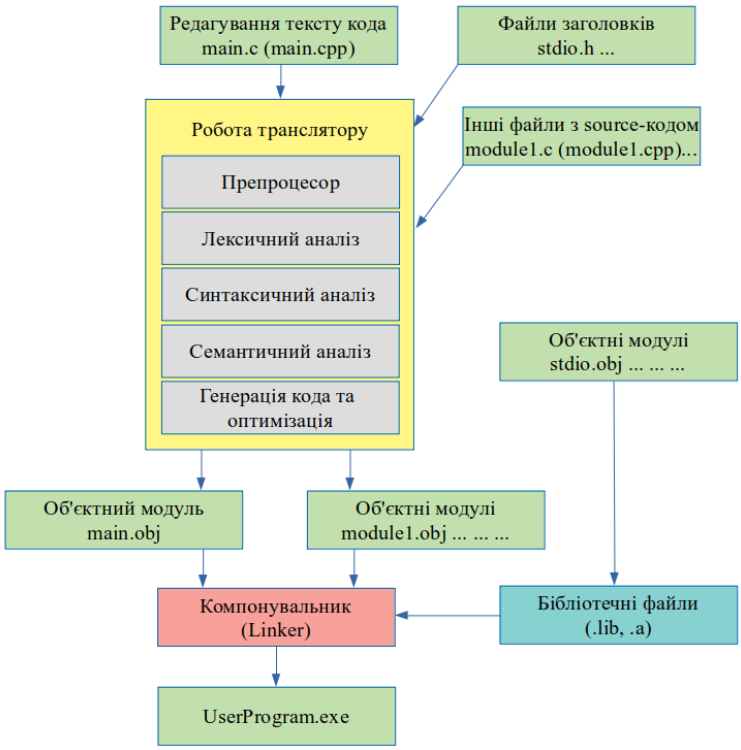


Рис. 3.5. Фази підготовки програми на мові C/C++

Трансляція включає в себе кілька фаз: препроцесор, лексичний, синтаксичний та семантичний аналіз, генерація коду та його оптимізація.

Препроцесор – попередня (нульова) фаза трансляції (компіляції) на рівні перетворення вихідного тексту програми.

Те, що повинен виконати препроцесор в програмі на С оформлено у вигляді так званих директив – рядків, які починаються з символу #.

Фактично препроцесор здійснює заміну одних частин програмного коду на інші, керує остаточним варіантом вихідного коду, який буде підданий подальшій обробці транслятором (компілятором).

При програмуванні на різних мовах подібні інструменти можуть називатися макропроцесором.

Лексика мови програмування – це правила "правопису слів" програми, таких як ідентифікатори, константи, службові слова, коментарі. *Лексичний аналіз* розбиває текст програми на зазначені елементи. Особливість лексики – її елементи являють собою регулярні лінійні послідовності символів. Наприклад, ідентифікатор – це довільна послідовність літер, цифр і символу "_", що починається з букви або "_" (але не з цифри!)

Синтаксис мови програмування – це правила складання речень мови з окремих слів. Такими реченнями є операції, оператори, визначення функцій та змінних. Особливістю синтаксису є принцип вкладеності (рекурсивність) правил побудови речень. Це означає, що елемент синтаксису мови в своєму визначенні прямо або побічно в одній з його частин містить сам себе. Наприклад, у визначенні оператора циклу тілом циклу є оператор, окремим випадком якого може бути все той же оператор циклу.

Семантика мови програмування – це сенс, який закладається в кожну конструкцію мови. *Семантичний аналіз* – це перевірка смислової правильності конструкції. Наприклад, якщо у виразі використовується змінна, то вона повинна бути визначена раніше за текстом програми, а з цього визначення може бути отриманий її тип. Виходячи з типу змінної, можна говорити про допустимість операції з цією змінної.

Генерація коду – це перетворення елементарних дій, отриманих в результаті лексичного, синтаксичного та семантичного аналізів програми, в деяке внутрішнє представлення. Це можуть бути коди команд, адреси та вміст пам'яті даних, або текст програми на мові Асемблеру, або стандартизований проміжний код. В процесі генерації коду проводиться і його оптимізація, наприклад, для скорочення розміру вихідного модуля або для підвищення швидкодії та ін.

Отриманий в результаті трансляції *об'єктний модуль* включає в себе готові до виконання коди команд, адреси та вміст пам'яті даних. Але це стосується тільки власних внутрішніх об'єктів програми (функцій та змінних).

Звернення до зовнішніх функцій і змінним, відсутнім в даному фрагменті програми, не може бути повністю переведено у внутрішнє представлення та залишається в об'єктному модулі у вхідному (текстовому) вигляді. Але, якщо ці функції та змінні відсутні, значить вони повинні бути якимось чином отримані в інших об'єктних модулях, які пройшли трансляцію раніше.

Це і є принцип *модульного програмування* – уявлення тексту програми у вигляді декількох файлів, кожен з яких транслюється окремо.

З модульним програмуванням стикаються в двох випадках.

1. Коли програміст сам пише модульну програму.
2. Коли програміст використовує стандартні бібліотечні функції.

Зовнішнє посилання – звернення до змінної або виклик функції у внутрішньому поданні модуля, які визначені в іншому модулі та відсутні в поточному.

Точка входу – адреса змінної або функції у внутрішньому поданні модуля, до яких можливе звернення з інших модулів.

Бібліотека об'єктних модулів – це файл (бібліотечний файл), що містить набір об'єктних модулів та власний внутрішній каталог. Об'єктні модулі бібліотеки витягуються з неї цілком при наявності в них необхідних зовнішніх функцій та змінних і використовуються в процесі компонування програми.

Компонування – це процес складання програми з об'єктних модулів, в якому проводиться їх об'єднання в програму для виконання і зв'язування викликів зовнішніх функцій і їх внутрішнього подання (кодів), розташованих в різних об'єктних модулях, точок входу. За компонування відповідає програма, яка називається компонувальником або редактором зв'язків, або лінковщиком, або скорочено – лінкером.

Алфавіт мови, символи

Під елементами мови розуміються його базові конструкції, які використовуються при написанні програм. Програма на мові C/C++ – послідовність символів, які збираються в лексеми, що включають базовий словник мови.

Лексема – це одиниця тексту програми, яка має самостійний сенс для компілятора мови і яка не містить в собі інших лексем. Існують п'ять категорій лексем:

• ключові слова;
• ідентифікатори;
• константи;
• оператори;
• знаки пунктуації.

Безліч символів мови С включає великі та малі літери латинського алфавіту та арабські цифри:

A, B, C, D, ... Z
a, b, c, d, ... z
0, 1, 2, 3, ... 9

Букви та цифри використовуються при формуванні констант, ідентифікаторів і ключових слів.

Компілятор мови С/С++ розглядає одну і ту ж прописну та малі літери як різні символи, наприклад: Cursor, cursor, CURSOR – три різних змінних.

Символи, що відокремлюють лексеми одну від одній, наприклад константи та ідентифікатори, складають групу символів пропусків. До цієї групи належать символи:

• пропуск;
• табуляція;
• переведення рядку;
• повернення каретки;
• нова сторінка;
• вертикальна табуляція;
• новий рядок.

Коментар – це послідовність символів, яка сприймається компілятором мови С/С++ як окремий символ пробілу та ігнорується.

Приклади коментарів в мові С/С++:

```
/ * Це коментар в один рядок * /

/ *  Це коментар
     в кілька
     рядків
* /
```

Інший приклад однорядкового коментаря в мові С++:

```
// Це коментар в один рядок
```

Багаторядкові коментарі не можуть бути вкладеними. Тобто не можна писати в коді таким чином:

```
/*
        Один
        коментар
  /*
        Інший, вкладений
        коментар
   */
   . . .
*/
```

Але однорядкові коментарі в стилі C++ (//) можуть бути вкладені в багаторядкові коментарі.

Символи-роздільники в мові С, мають спеціальний сенс для компілятора мови:

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

Спеціальні символи (спецсимволи) призначені для подання пропусків та неграфічних символів (наприклад, звуковий сигнал динаміка комп'ютера) в рядках та символьних константах. Спецсимвол складається зі зворотного слешу (\ ), за яким йде або буква, або знаки пунктуації, або комбінація цифр. Спецсимволи мови С/С++ наведені в таблиці 3.1.

При роботі з простим текстовим редактором (Notepad, Edit та їм подібним) введення кожного рядка завершується натисканням клавіші ENTER (ВВЕДЕННЯ). Фактично при цьому в ОС MS-DOS або MS Windows в текст вставляються два символи: повернення каретки та новий рядок (коди в шістнадцятковій системі: 0D 0A); в Linux-сумісних ОС – це символ новий рядок (0A); в старих версіях Mac OS – це символ повернення каретки (0D). В новітніх версіях macOS – це символ з кодом 0A. Однак стандартні бібліотечні функції введення/виведення (I/O) текстової інформації розглядають цю пару символів як один символ – символ нового рядка (0A). Цей символ представляється в символьних константах та символьних рядках як – '\n'. При читанні текстового рядка стандартні бібліотечні функції замінюють згадану пару символів єдиним символом нового рядка, а під час запису символу нового рядка додають перед ним символ повернення каретки (для ОС Microsoft).

Таблиця 3.1

Спеціальні символи мови С/С++

Спеціальний символ	*Шістнадцяткове значення в коді ASCII*	*Найменування*
\n	A	Новий рядок
\t	9	Горизонтальна табуляція

\v	B	Вертикальна табуляція
\b	8	Забій, повернення на символ назад (Backspace)
\r	D	Повернення каретки (CR)
\f	C	Прогін сторінки
\a	7	Звуковий сигнал
\'	2C	Апостроф
\"	22	Подвійна лапка
\\	5C	Зворотній слеш
\?	3F	Знак питання
\0		Кінець С-рядку
\0dd		Байтове значення у вісімковому представленні
\xdd (\Xdd)		Байтове значення у шістнадцятковому представленні

Конструкція – \ddd – дозволяє задати довільне байтове значення як послідовність від однієї до трьох вісімкових цифр. Конструкція – \xdd або \Xdd – дозволяє задати довільне байтове значення як послідовність від однієї до двох шістнадцятирічних цифр. Наприклад, символ забій в коді ASCII може бути заданий як \010 або \x08 , або \x8.

Крім спецсимволів, зворотній слеш використовується також як ознака продовження символьних рядків та макровизначень препроцесора.

Ідентифікатори – це імена змінних, функцій та міток, які використовуються в програмі. Першим повинен стояти символ ‘a’ – ‘z’, ‘A’ – ‘Z’ або знак підкреслення. Далі можуть бути будь-які символи, цифри або підкреслення. Для дотримання гарного стилю програмування слід вибирати зрозумілі імена ідентифікаторів, які полегшують документування програм. Слід уникати використання підкреслення як початкового символу, так як його використовують багато системних програм та бібліотек.

Ключові слова в мові С

Ключові слова – це зарезервовані ідентифікатори, які мають спеціальне значення для компілятора мови. Імена програмних об'єктів програми (наприклад, імена змінних) не можуть співпадати з ключовими словами.

В різних стандартах та діалектах мови С список ключових слів може бути розширений.

Переглянути опис таких ключових слів можна, наприклад, за посиланням:

https://en.cppreference.com/w/c/keyword

Основні ключові слова мови С (стандарт С99):

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

Константи

Константа в С/С++ – це ціле число, число з рухомою (плаваючею) точкою, символ або рядок символів. Константа може бути довгою та беззнаковою.

Приклади констант:

Десяткові константи	Вісімкові константи	Шістнадцятирічні константи
10	012	0xa або 0xA
132	0204	0x84
32179	076663	0x7db3 або 0x7DB3

Символьна константа – це буква, цифра, знак пунктуації або спеціальний символ, укладений в апострофи.

Приклади:

'a' '\' '\\' '\n' '3'

Рядкова константа, рядковий літерал, або просто літерал – це послідовність символів, яка укладена в подвійні лапки. Символьний рядок розглядається як масив символів, кожний елемент якого представляє окремий символ. У внутрішньому представленні рядка в кінці присутній символ '\0', так що фізичний обсяг пам'яті для зберігання рядка перевищує кількість символів, записаних між лапками, на одиницю. Такі рядки отримали назву С-рядків. Функції, по роботі з рядками, повинні враховувати цю особливість. Бібліотечна

функція, яка повертає довжину рядка, повертає значення довжини без урахування символу '\0'. Цей символ ‘\0’ є дуже корисним при обробці рядків, оскільки по його наявності програміст знає, що рядок закінчився. Тобто це ознака закінчення рядка.

Операції

Операції – це комбінації символів, що визначають дії по перетворенню значень. Компілятор мови інтерпретує кожну з цих комбінацій як самостійну лексему.

Таблиця 3.2

Операції в мові програмування С

Операція	*Найменування*	*Операція*	*Найменування*
!	Логічне заперечення, НІ	==	Дорівнює
~	Зворотній код (побітова інверсія)	!=	Не дорівнює
+	Додавання; унарний плюс	&	Порозрядне І; адресація
–	Віднімання; унарний мінус	|	Порозрядне АБО
*	Множення; непряма адресація	^	Порозрядне виключаюче АБО (хог)
/	Ділення	&&	Логічне І
%	Залишок від ділення цілих чисел	||	Логічне АБО
<<	Побітове зміщення вліво	,	Послідовне виконання
>>	Побітове зміщення вправо	?:	Умовна операція
<	Менше	++	Інкремент
<=	Менше або дорівнює	--	Декремент
>	Більше	=	Просте присвоювання
>=	Більше або дорівнює	+=	Присвоєння з додаванням
*=	Присвоєння з множенням	–=	Присвоєння з відніманням
/=	Присвоєння з діленням	%=	Присвоєння з залишком від ділення
>>=	Присвоєння зі зміщенням вправо	<<=	Присвоєння зі зміщенням вліво
&=	Присвоєння з порозрядним І	|=	Присвоєння з порозрядним АБО
^=	Присвоєння з порозрядним виключаючим АБО (хог)		

Умовна операція *?:* є не двохопérандною, а тернарною (з трьома операндами) операцією.

Приклади:

a = 5; b = 2; a++; a = a+1; --b; b--; c = (a>b)a:b;

a+= 10 тож саме, що: a = a+10;

Якщо в одному виразі комбінується декілька операцій, то потрібно пам’ятати про пріоритет виконання (таблиця 3.3). Операції з вищими пріоритетами обчислюються першими. Найвищим пріоритетом є пріоритет рівний 1.

Таблиця 3.3

Пріоритет операцій

Пріоритет	*Знак операції*	*Найменування*	*Порядок виконання*
1	() [] . ->	Первинні	Зліва направо
2	- ~ ! * & ++ -- sizeof	Унарні	Справа наліво
3	* / %	Мультиплікативні	Зліва направо
4	+ -	Адитивні	
5	<< >>	Зсув	
6	< > <= >=	Відношення	
7	== !=	Відношення (рівність)	
8	&	Порозрядне І	
9	^	Порозрядне виключаюче АБО	
10	|	Порозрядне АБО	
11	&&	Логічне І	
12	||	Логічне АБО	
13	?:	Умовна	
14	= *= /= %= += -= <<= >>= &= |= ^=	Просте та складне присвоювання	Справа наліво
15	,	Послідовне обчислення	Зліва направо

Виділяють дві форми запису інкременту та декременту: префіксна та постфіксна, які відіграють роль у складних виразах:

– префіксна: ++a; --a;

– постфіксна: a++; a--;

Приклади, що показують різницю:

data[i++] = 5; призведе до наступних дій: data[i]=5; i = i+1;

data[++i] = 5; призведе до наступних дій: i = i+1; data[i] = 5;

Базові типи даних

Базові типи даних представлені в таблиці 3.4.

Тип *void* (порожній) має спеціальне призначення. Вказівка специфікації типу *void* в оголошенні функції означає, що функція не повертає значень. Вказівка типу *void* у списку оголошень аргументів в оголошенні функції означає, що функція не приймає аргументів. Можна оголосити покажчик на тип

void; він буде вказувати на будь-який, тобто неспецифікований тип. *Не можна оголосити змінну типу void!*

Окрім наведених в таблиці 3.4, в залежності від стандарту С, можливі використання й інших типів. Наприклад, long long, _Bool, _Complex та ін. Переглянути опис таких типів даних можна, наприклад, за посиланнями:

https://en.cppreference.com/w/c/keyword
https://en.cppreference.com/w/c/language/arithmetic_types

Таблиця 3.4

Базові типи даних

Базові типи	*Специфікація типів*		
Цілі	**signed char** (1 байт)	**char**	от -128 до 127
	signed int (4 байти)	**int**	от -2147483648 до 2147483647
	signed short int (2 байти)	**short** **signed short**	от -32768 до 32767
	signed long int (*)	**long** **signed long**	от -9223372036854775808 до 9223372036854775807
	unsigned char (1 байт)	–	от 0 до 255
	unsigned int (4 байти)	**unsigned**	от 0 до 4294967295
	unsigned short int (2 байти)	**unsigned short**	от 0 до 65535
	unsigned long int (*)	**unsigned long**	от 0 до 18446744073709551615
	wchar_t (**)	–	от 0 до 65535 (для 2-х байт)
Плаваючи	**float** (4 байти)	Плаваючий одинарної точності	Стандартний формат IEEE
	double (8 байт)	Плаваючий подвійної точності	
	long double (***)	Довгий плаваючий подвійної точності	
Інші	**void**	Порожній	–
	enum	Перелічувальний	

* – залежить від архітектури системи, найбільш часто становить або 4 або 8 байт;

** – залежить від версії компілятора та платформи; введений в стандарті ANSI/ISO C; може мати різний розмір, наприклад, 2 або 4 байта.

*** – залежить від архітектури системи, компілятора і найбільш часто становить 8, 10, 12 або 16 байт.

Приклади оголошень змінних різних типів:

```
int a;
float b, c;
double d;
char ch;
unsigned char uCh;
```

Структура програми на С та її збірка в різних середовищах

Мінімальна програма на мові С може бути представлена в наступному вигляді (source-файл ex1.cpp):

```
int main( )             // заголовок визначення функції main
{                       // початок блоку операторів
        return 0;       // оператор повернення значення з функції
}                       // кінець операторного блоку
```

Найбільш відомі C/C++ компілятори: GNU GCC, Microsoft CL, Clang. Найбільш відомі інтегровані середовища розробки – IDE (*Integrated Development Environment*) для розробки на C/C++: MS Visual Studio, Qt Creator, Code::Blocks, C++ Builder, KDevelop, Eclipse, NetBeans, CLion.

Приклад консольної команди збірки в середовищі Unix-сумісної ОС та компілятора GCC:

```
gcc -o ./ex1 ./ex1.c
```

або

```
g++ -o ./ex1 ./ex1.cpp
```

У найпростішому випадку за опцією -o вказують ім'я файлу для виконання. Далі вводять ім'я файлу вхідного коду.

Збірка в MS Windows за допомогою компілятора командного рядку CL в разі встановленого пакета MS Windows SDK або MS Visual Studio (в системній змінній *PATH* повинен бути вказаний шлях до розміщення компілятору *cl*):

```
cl ex1.c
```

Для збірки програми у різних IDE, використовують елементи графічного інтерфейсу користувача (GUI – *Graphical User Interface*) цих програм – пункти меню та діалогові вікна. Наприклад, для збірки проекту в середовищі Code::Blocks версії 20.03 потрібно виконати наступні кроки:

1. Вибрати пункт меню *File \ New \ Project*.

2. У діалоговому вікні з категорії проектів вибрати *Console* та далі *Console application*. Натиснути на кнопку *Go*, потім на *Next*.

3. У наступному діалозі вказати мову C++, потім *Next*.

4. Далі в діалозі вказати *Project title* (ім'я програми), наприклад, Hello, а також *Folder to create project in* (папку для розміщення проекту). Наприклад, диск Z:\projects та натиснути на *Next*.

5. У фінальному діалозі нажати на кнопку *Finish*.

Для збірки проекту використовувати пункт меню *Build \ Rebuild*. Для запуску – *Build \ Run* (Ctrl + F10 або F9).

Для збереження змін, внесених до тексту коду, використовувати пункт меню *File \ Save file* (Ctrl + S) або подібний.

Основною точкою входу в програму є функція *main( )*. Її заголовок може бути різним, залежно від задачі, що вирішується. Наприклад, найбільш часто використовують наступні варіанти заголовка *main( )*:

```
int main( )

void main( )

// функція main( ) з параметрами:
int main( int argc, char *argv[] )

// функція main( ) з параметрами:
int main( int argc, char *argv[], char *envp[] )
```

Як правило параметри *argc* та *argv* призначені для обробки даних, які надходять до програми у вигляді аргументів, що передані програмі через командний рядок.

Параметр *envp* використовується тільки у випадках, коли програмі необхідно обробити дані, які зберігаються в так званих змінних середовища оточення ОС, що можуть сильно відрізнятися для різних ОС.

Функція *main( )* може повернути код завершення програми в ОС тільки як ціле значення зі знаком (тип повернення – int).

Можливий варіант заголовка функції *main( )* з типом повернення *void* – тобто функція нічого не повертає. Однак, не всі версії певних компіляторів нормально сприймають такий заголовок.

Варіант заголовка *int main(void)* – такий же, як і *int main( )*, тобто параметри функції відсутні.

Вважається доброю практикою використання заголовку функції *main( )* з поверненням значення цілого типу.

Основні директиви препроцесора

Препроцесор мови C/C++ виконує макропідстановки, умовну компіляцію та включення іменованих файлів. Будь-яка директива препроцесора C/C++ починається з символу #. Наприклад:

```
#include  <stdio.h>
#include  "mylib.h"
#include  "..\..\mylib\include\main_mylib.h"
#include  "../../mylib/include/main_mylib.h"
#include  "c:\projects\mylib\include\main_mylib.h"
```

Директива препроцесора *include* призводить до вставки всього вмісту зазначеного файлу в точку коду, де вона вказана. Найчастіше це так звані файли заголовків (header-файли, h-файли). Такі файли мають, як правило розширення .h, .hpp, .hh або взагалі не мають розширення, наприклад, в стандартній бібліотеці C++.

У header-файлах, як правило, можуть знаходитися оголошення різних функцій, нові типи даних, посилання на зовнішні глобальні змінні, які оголошені глобально, а також директиви препроцесора. Програмний код в таких файлах відсутній. Винятки становлять h-файли з описом класів-шаблонів в C++, які можуть містити код. У всіх інших випадках, як правило, код зберігається в файлах з розширенням .c, .cpp, .cc.

Перший варіант *include* з кутовими дужками є найбільш поширеним. Він призначений для включення header-файлів, розміщених в певних для середовищ розробки або для компіляторів, каталогах. Ці каталоги, як правило, іменуються як *include*. Наприклад:

```
/usr/include                    – в Unix-подібних ОС
C:\Program Files\….\include     – у випадку з встановленим пакетом MS Visual Studio
```

Другий варіант *include* зазвичай використовують у випадках, коли треба включити h-файл, розташований в тому ж каталозі проекту, що і файл вихідного коду. Якщо h-файл розташований в іншому місці, то застосовують відносний шлях до файлу (наприклад, 3-й варіант для Windows, 4-й - для Unix-подібних).

Представлений 5-й варіант використовують рідко, оскільки в ньому використаний абсолютний шлях. Вказівка абсолютного шляху є поганою практикою в програмуванні!

Інша директива препроцесору, яка широко використовується, це *define*, представлена в трьох варіантах:

#define ідентифікатор
#define ідентифікатор текст
#define ідентифікатор(список_параметрів) текст

Ця директива замінює всі входження ідентифікатора у вхідному файлі на текст, який слідує за цим ідентифікатором. Якщо за ідентифікатором йде список параметрів, то директива стає макровизначеннями з аргументами. Текст являє собою набір лексем, таких як ключові слова, константи, ідентифікатори або вираз. Якщо вказано тільки ідентифікатор – то в цьому випадку він просто стає відомий для препроцесора (перший варіант). Як правило, цей варіант необхідний, якщо такий ідентифікатор буде використаний далі в інших директивах препроцесора. Наприклад, в директивах перевірки умови.

Приклади:

```
#define __UNIX_TYPE__
#define MAX  11
#define WIDTH  80
#define LENGTH (WIDTH+10)
#define FILEMESSAGE "Error! File Not Found\n"
#define MULT(a, b) ((a)*(b))
```

В останньому випадку представлено макровизначення з параметрами. Кількість дужок відіграє велику роль. Наприклад, якщо дано наступний варіант макровизначення:

```
#define MULT(a, b) (a*b)
```

то в місці макровиклику a = MULT (3 + 4, 5 + 6) отримаємо результат: a = (3 + 4 * 5 + 6), який буде невірним! У попередньому випадку отримуємо вірний результат: a = ((3 + 4) * (5 + 6)).

Директива *undef* говорить препроцесору «забути» визначення ідентифікатору, виконане раніше. У разі, якщо зазначений ідентифікатор був раніше невизначений, це не вважається помилкою. Наприклад:

```
#undef MULT
```

Зв'язка директив *ifdef-else-endif*, *ifndef-else-endif*, *ifdef-endif*, *ifndef-endif* дозволяє виконувати так звану умовну компіляцію. Тобто залишати у вхідному коді для компілятора мови тільки ту частину, яка відповідає деякій умові. Наприклад, якщо визначений ідентифікатор _DEBUG, що говорить про режим збірки з відлагодженням, то в код буде вбудований виклик певної функції:

```
#ifdef _DEBUG
     print_debug_message( ERR_MSG_1 );
#endif
```

Інший приклад:

```
#ifdef MULT
     a = MULT(c + d,  e + f);
#else
     a = (c+d)*(e+f);
#endif
```

Найпростіший консольний друк інформації

Для виведення інформації на консоль в С використовуються функції, оголошення яких надані в файлі ***stdio.h*** (*Standard Input/Output*). Наприклад, виведення рядку тексту на консоль з переведенням курсору на новий рядок, здійснюється функцією *puts( )*. Для виведення символу на консоль використовується функція *putchar( )*:

```
puts("Hello!");
putchar('a'); putchar('\n');
```

Широкі можливості дає форматований вивід (виведення за форматом) на консоль за допомогою функції *printf( )*. Функція приймає змінне число аргументом з першим обов'язковим параметром у вигляді рядка. Наприклад:

```
printf("Hello!\n");
```

У рядку можуть бути використаний специфікатор формату для типу, наприклад, такий як:

%d, %i – виведення цілого;

%x, %X – виведення цілого в шістнадцятковій системі числення;

%o – виведення цілого в вісімковій системі числення;

%c – виведення символу;

%s – виведення рядку символів;

`%ls` – виведення Unicode-рядку символів;

%f – виведення дійсного;

%e, %E – виведення дійсного в Е-нотації (decimal exponent notation);

`%lf` – виведення довгого дійсного подвійної точності.

Повний список та опис специфікаторів формату можна подивитися за посиланням: https://en.cppreference.com/w/c/io/fprintf

Наприклад:

```
int a = 5;
int b = 7;
printf( "A + B = %d\n",  a+b  );
```

або

```
int a = 5, b = 7, c = 0;
c = a+b;
printf( "A + B = %d\n",  c );
printf( "%d + %d = %d\n",  a, b, c );
printf( "A + B = %d\t -> %d + %d = %d\n",  c, a, b, c );
```

Завдання. Необхідно написати на мові С програму, яка обчислює вираз: $y = \sin^2(x^3)$, при x = 3,7.

Рішення:

```
#include <stdio.h>
#include <math.h>
int main()
{
     // Оголошуємо змінну та присвоюємо їй значення 3.7
     double x = 3.7;

     // Виконуємо розрахунки математичного виразу
     double y = pow( sin( pow(x, 3.0) ), 2.0 );
     // Це еквівалентно:
     // double y = sin( x*x*x )*sin( x*x*x );

     // Виводимо на консоль результат
     printf("Y=%lf\n", y);

     return 0;
}
```

Результат виконання програми:

Y=0.142802

Де *pow( )* та *sin( )* – математичні функції, заголовки яких містяться в *math.h:*

double pow(double x, double y); – *функція* x^y
double sin(double x); – *функція sin(x)*

Лекція 4. Оператори мови С

Список операторів мови С:

Порожній оператор	;
Складений оператор або блок	{ }
Оператор-вираз	<вираз>;
Умовний оператор	if
Оператор покрокового циклу	for
Оператор циклу з передумовою	while
Оператор циклу з постумовою	do
Оператор продовження	continue
Оператор розриву	break
Оператор-перемикач (оператор множинного вибору)	switch
Оператор повернення	return
Оператор переходу (вкрай не рекомендується використовувати)	goto

Порожній оператор

Це оператор, що складається тільки з точки з комою. Він може розташовуватися в будь-якому місці коду програми, де за правилами синтаксису може бути оператор. Виконання порожнього оператора не міняє стану програми.

Складений оператор (операторний блок, блок операторів)

```
{
        [<оголошення>]
        . . .
        . . .
        . . .
        [<оператор>]
}
```

У складеному операторі можуть міститися оголошення та визначення змінних, локальних для даного блоку, а також різні оператори. Якщо в такому

блоці оголошується звичайна (нестатична) змінна, то її область видимості і час життя буде обмежено цим блоком.

Синтаксис умовного оператора if

```
if(<вираз>)
        <оператор_1>
[ else
        <оператор_2> ]
```

Приклади:

```
if(i>0) y=x/i;
else {
        x=i;
        y=f(x);
}
```

Оператор *if* може бути вкладений в <оператор_1> або <оператор_2> іншого оператора *if*. При вкладенні операторів *if* рекомендується для ясності групування операторів використовувати фігурні дужки, що обмежують <оператор_1> та <оператор_2>. Якщо ж фігурні дужки відсутні, то компілятор асоціює кожне ключове слово *else* з найближчим оператором *if*, у якого відсутня конструкція *else*:

```
if(i>0)                 if(i>0) {
  if(j>i) x=j;            if(j>i) x=j;
  else x=i;               }
                          else x=i;
```

Синтаксис оператора циклу з передумовою while

Тіло оператора циклу *while* виконується до тих пір, поки <умовний_вираз> не стане хибним.

```
while(<умовний_вираз>)
  <оператор>
```

Приклад:

```
i=0;
while(i<10) {
        printf(“i – %d\n”, i);
        i++;
}
```

Синтаксис оператора циклу з постумовою do-while

Тіло оператора циклу do-while виконується до тих пір, поки <умовний_вираз> не стане хибним.

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

Приклад:

```
i=0;
do {
        printf(“i – %d\n”, i);
        i++;
}while(i<10);
```

Синтаксис оператора покрокового циклу for

```
for(  [<початковий_вираз>] ;
      [<вираз_умови>] ;
      [<вираз_збільшення_або_зменшення>] )
  <оператор>
```

Тіло оператора циклу for виконується до тих пір, поки <вираз_умови> не стане хибним. <початковий_вираз> та <вираз_збільшення_або_зменшення> зазвичай використовуються для ініціалізації та модифікації параметрів циклу або інших значень. Якщо <вираз_умови> відсутній, то його значення приймається за істину – цикл *for* буде нескінченним. Приклад:

```
for(i=space=tab=0; line[i] != '\0'; i++) {
        if(line[i]==’\x20’) space++;
        if(line[i]==’\t’)     { tab++; line[i]=’\x20’; }
}
```

У цьому прикладі підраховуються символи пробілу (код '\x20') та горизонтальної табуляції ('\t') в масиві символів з ім'ям *line* і проводиться заміна кожного символу горизонтальної табуляції на пробіл. Тіло *for* виконується до тих пір, поки не буде досягнутий кінець рядка (символ кінця С-рядку – '\0').

Оператори continue та break

Оператор *continue* може перебувати тільки всередині тіла будь-якого з трьох циклів. Він виконує перехід до наступної ітерації самого внутрішнього циклу, що містить його. Оператор *break* використовується в тілі будь-якого оператора циклу, а також в тілі оператора множинного вибору *switch*.

Наприклад:

```
int i = 0;
while( i<MAX )
{
    if( a[i] == findValue) { findIndex = i; break; }
    i++;
}

i = 10;
while( i>0 )
{
   x = f(i);
   i--;
   if( 1 == x ) continue;
   y += (x*x);
}
```

Оператор-перемикач switch (оператор множинного вибору)

```
switch( <вираз> )
{
        [<оголошення>]
        [case <константний-вираз>:]  [<оператор>]
        . . .
        [case <константний-вираз>:]  [<оператор>]
        [default:] [<оператор>]
}
```

Цей оператор призначений для вибору одного з декількох альтернативних шляхів виконання програми. Виконання оператора-перемикача починається з обчислення значення виразу, що стоїть в круглих дужках. Після цього управління передається одному з операторів тіла перемикача. За ключовим словом *case* йде константа варіанту, яка порівнюється зі значенням виразу *switch*. Якщо жодна з case-констант не збіглася, то виконується default-частина. Значенням switch-виразу повинна бути величина цілого типу (у класичній версії мови програмування С). Наприклад:

```
char ch = 'a';
. . .
switch(ch)
{
        case 'A': capa++; break;
        case 'a':  lettera++; break;
        default:  total++;
}

#include <ctype.h>
. . .
switch(ch)
{
        case 'a': case 'b': case 'c': case 'd': case 'e': case 'f': ch = toupper(ch);
}
```

Лекція 5. Покажчики, масиви даних, динамічна пам’ять, функції, консольне введення, константи

Покажчики

Типова комп'ютерна система містить масив послідовно пронумерованих (адресованих) комірок пам'яті, з якими можна працювати окремо або цілими безперервними групами.

Покажчик – це змінна, яка містить адресу іншої змінної. Наприклад:

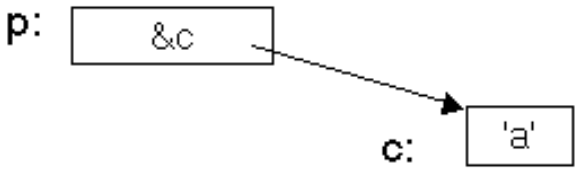


Рис. 5.1. Представлення покажчика

Рис. 5.1 відображає наступну ситуацію:

```
char c = 'a';          // оголошення та визначення змінної c
char* p = &c;          // p – покажчик на тип char, який містить адрес
                       // комірка пам’яті змінної c
```

Одномісна (унарна) операція & в даному контексті дає адресу об'єкту – адреса змінної *c*. Після виконання цієї операції кажуть, що *p* вказує на *c*. Цю операцію застосовують лише до об'єктів, що зберігаються в оперативній пам'яті – у змінних або елементах масиву.

Одномісна операція * називається операцією посилання за вказівником або розіменуванням. Застосовуючи її до покажчика, отримують об'єкт, на який він вказує. Наприклад:

```
int x = 10;
int y = 0;
int * px = &x;       // оголошення вказівника px та ініціалізація його
                     // адресою змінній x
int * px2;           // оголошення вказівника px2
x++;          // x = 11,  *px = 11
(*px)++;      // x = 12,  *px = 12
y = *px;      // y = 12
```

```
++*px;        // x = 13,  *px = 13
px2 = px;     // адреса px2 та px – рівні, тобто *px2 = 13
px2 = &y;     // px2 буде містити адресу y: *px2 = 12
```

Гарантується, що немає об'єктів з нульовою адресою. Отже, покажчик, що дорівнює нулю (0), можна інтерпретувати як покажчик, який ні на що не посилається. Для встановлення цього факту в мові C/C++ використовують макроідентифікатор NULL. Наприклад:

```
int * p = NULL;  // p – ні на що не посилається.
```

NULL можна використовувати всюди у виразах де потрібна константа та потрібен покажчик з нульовою адресою.

Масиви даних

Масив – спеціальна структура даних, що дозволяє зберігати як єдине ціле послідовність змінних однакового типу. Оголошення масиву визначає тип даних всіх його елементів та ім'я масиву. В оголошенні також може бути вказана кількість елементів в масиві. Всі елементи масиву розташовані в послідовно розташованих один за одним комірках пам'яті.

Масиви можуть бути одновимірними та багатовимірними – двовимірними, тривимірними і т.д. Часто кажуть, що масив є багатовимірним, коли мірність масиву вже перевищує 3-х.

Одновимірний масив являє собою вектор даних. Двовимірний – матрицю даних. Тривимірний – куб даних.

Елементи багатовимірного масиву зберігаються порядково. Наприклад, якщо є двовимірний масив з двох рядків та трьох стовпців, то спочатку в пам'яті буде зберігатися три елементи першого рядка, потім три елементи другого рядка.

Приклад оголошення та визначення одновимірного масиву даних цілого типу (вектора даних) з п'яти елементів:

```
int data1[5];                          // оголошення масиву
int data1[5] = {1, 2, 33, 47, 15}; // оголошення та визначення масиву
int data1[] = {1, 2, 33, 47, 15};   // оголошення та визначення масиву
```

Приклад оголошення та визначення двовимірного масиву (два рядки та чотири стовпці):

```
int data2[2][4];
int data2[2][4] = { {1, 2, 3, 4}, {5, 6, 7, 8} };
```

int data2[][4] = { {1, 2, 3, 4}, {5, 6, 7, 8} };

Приклад оголошення та визначення тривимірного масиву:

```
int data3[2][4][3];
int data3[2][4][3] = { { {1,   2,   3},
                         {4,   5,   6},
                         {7,   8,   9},
                         {10, 11, 12} },
                       { {13, 14, 15},
                         {16, 17, 18},
                         {19, 20, 21},
                         {22, 23, 24} }
                     };
```

Індексація елементів масивів в C/C++ починається завжди з нуля. Якщо кількість елементів масиву визначена, то його, як правило, краще описати через макроідентифікатор. Наприклад:

```
. . .
#define MAX 10
. . .
int data1[MAX];
. . .
```

У мові С немає вбудованого типу даних, для операції з рядками. Рядки в С – це одномірні масиви символів. За правилами мови С, кожний С-рядок повинен завершуватися символом '\0'. Наприклад:

```
char str[] = "This is test";
```

В цьому випадку символ '\0' присутній неявно при ініціалізації.

Для звернення до елементів масивів використовують оператори циклів. Наприклад:

```
. . .
for( int i=0; str[i] != '\0'; i++ ) putchar( str[i] );  // посимвольний вивід
putchar('\n');
. . .
for(int i=0; i<MAX; i++) data1[i] = 0; // обнулення елементів вектору data1
. . .
for( int i=0; i<2; i++ ) // присвоєння елементу матриці (2 рядки, 4 стовпці)
   for( int j=0; j<4; j++ ) data2[i][j] = i*j;
. . .
for(int i=0; i<2; i++) // виведення значення тривимірного масиву
{
    for(int j=0; j<4; j++)
    {
     for(int k=0; k<3; k++) printf( "data[%d][%d][%d] = %d ", i, j, k, data[i][j][k] );
     printf("\n");
```

```
    }
    printf("\n\n");
}
. . .
```

Між масивом та покажчиком в C/C++ існує тісний зв'язок – ім'я масиву завжди вказує на перший елемент масиву. Наприклад, змінюємо адресу покажчика *p* в циклі для переміщення на наступний елемент вектору:

```
int * p = data1;
for(int i=0; i<5; i++, p++) printf("%d\n", *p);
p = &data1[0];
for(int i=0; i<5; i++, p++) printf("%d\n", *p);
```

У разі багатовимірного динамічного масиву та звичайного багатовимірного масиву є відмінність – вони зберігаються в пам'яті по різному в силу специфіки виділення пам'яті під багатовимірний динамічний масив. Тобто такі оголошення – не одне й те саме в плані подання до пам'яті:

```
int a[5][10];
int ** pa;
pa = a; // Помилка!
```

Динамічна пам’ять та покажчики

Динамічний розподіл пам'яті – це спосіб виділення оперативної пам'яті комп'ютера для об'єктів в програмі, при якому виділення пам'яті під об'єкт здійснюється під час виконання програми.

При динамічному розподілі пам'яті об'єкти розміщуються в так званої «купі» (англ.: heap). При конструюванні об'єкту вказується розмір, запитуваної під об'єкт пам'яті, і, в разі успіху, програма отримує адресу першої комірки виділеної пам'яті.

При звільненні зайнятої раніше під який-небудь об'єкт пам'яті, вона повертається в «купу» і стає доступною для операцій виділення пам'яті.

У міру створення в програмі нових об'єктів, кількість доступної пам'яті зменшується. Звідси випливає необхідність постійно звільняти раніше виділену пам'ять. В ідеальній ситуації програма повинна повністю звільнити всю пам'ять, яка потрібна була для роботи.

Некоректний розподіл пам'яті приводить до так званого «витоку» пам'яті, коли виділена пам'ять не звільняється. Багаторазові витоки пам'яті можуть призвести до вичерпання всієї оперативної пам'яті та порушення роботи ОС.

Для управління динамічною пам'яттю в мові С використовується ряд функцій: *malloc( ), calloc( ), realloc( ), free( )*. Приклад виділення пам'яті під значення цілого типу за допомогою функції *malloc( )*:

```
#include <stdlib.h>
. . .
int * pa = (int *)malloc( sizeof(int) );
```

Функція *malloc( )* виділяє пам'ять зазначеного розміру та привласнює адресу першої комірки виділеного блоку вказівником. При цьому використовується операція приведення типу, оскільки *malloc( )* повертає тип *void **.

Якщо пам'ять зазначеного розміру виділити не вдалося, то функція повертає NULL.

Якщо пам'ять була виділена за допомогою *malloc( )*, то вона повинна бути обов'язково звільнена та повернена в управління системою за допомогою функції *free( )*:

```
free( pa );
pa = NULL;
```

Приклад виділення пам’яті під одновимірний динамічний масив:

```
#define MAX 10

        p = (int *)malloc(sizeof(int)*MAX);
        if(NULL !=p )
        {
                for(int i=0; i<MAX; i++) { p[i] = i; printf("%d ", p[i]);}
                printf("\n");

                free(p); p = NULL;
        }
```

Приклад виділення пам’яті під двовимірний динамічний масив:

```
#define ROW             2
#define COL             3
int **pmatrix = NULL;

pmatrix = (int**) malloc( ROW * sizeof(int*) );
if(NULL!=pmatrix)
{
        for(int i=0; i<ROW; i++) pmatrix[i] = (int*) malloc( COL* sizeof(int) );

        for(int i=0; i<ROW; i++)
        {
                for(int j=0; j<COL; j++) { pmatrix[i][j] = (i+1)*(j+1); printf("%d ", pmatrix[i][j]);}
```

```
                printf("\n");
        }
        printf("\n");
        for(int i=0; i<ROW; i++) free(pmatrix[i]);

        free(pmatrix); pmatrix = 0;
}
```

Функції

Функція – це сукупність оголошень та операторів, призначена для виконання деякої окремої задачі.

Оголошення функції специфікує ім'я, формальні параметри та завжди закінчується символом ';'. Допускається відсутність імен формальних параметрів в оголошенні функції.

Визначення функції в програмі має бути тільки одне. Воно специфікує ім'я, формальні параметри та тіло функції.

Якщо функція не повертає в точку виклику значень, то тип повернення функції вказується *void*.

Приклади оголошення функцій:

```
int add( int x, int y );
int mult( int, int );
void print( char * str );
void swap( int * x, int * y );
```

Приклади визначення функцій:

```
int add( int x, int y ) {  return x+y; }

int mult( int a, int b) { return a+b; }

void print( char * str ) { puts( str ); }

void swap( int * x, int * y ) {  int tmp = *x;  *x = *y;  *y = tmp;  }
```

Приклади виклику функцій:

```
…
int a, b, c;
…
a = 5;  b = 10;
c = add( a, b );
…
swap( &a, &b );
…
```

Функцію можна визначити зі специфікатором *inline*. Такі функції називаються вбудованими. Наприклад, реалізація рекурсивної функції (викликає саму себе до певної умови):

inline int fact (int n) {return (n <2)? 1: n * fact (n-1); }

Специфікатор *inline* вказує компілятору, що він повинен намагатися кожного разу генерувати в місці виклику код, відповідний функції *fact( )* (факторіал), а не створювати окремо код функції (один раз) і потім викликати його за допомогою звичайного механізму виклику.

Структура оголошення та визначення функції в одномодульній програмі:

```
#include <stdio.h>
void print(const char * str);
int main( )
{
        print("This is test!\n");
        return 0;
}
void print(const char * str)
{
        printf( str );
}
```

У випадку багатомодульної програми:

```
//файл mylib.h
#ifndef __MYLIB_H__
#define __MYLIB_H__
void print( const char * str );
#endif

//файл mylib.cpp
#include <stdio.h>
#include "mylib.h"
void print(const char * str)
{
        printf( str );
}

//файл main.cpp
#include "mylib.h"
int main( )
{
        print( "This is test!\n" );
        return 0;
}
```

При використанні сучасних версій компіляторів GCC, CL, замість директив препроцесору #ifndef-#define-#endif можна скористатися однією директивою #pragma once. Наприклад:

```
// файл mylib.h
#pragma once
void print( const char * str );
```

Використання констант розглянуто далі.

Приклад збірки багатомодульної програми компілятором командного рядку GNU GCC:

```
g++ -o ./testprogram ./main.cpp ./mylib.cpp
```

Приклад збірки багатомодульної програми компілятором командного рядку Microsoft C++ Compiler:

```
cl main.cpp mylib.cpp /Fe:testprogram
```

Консольне введення даних

Для форматованого консольного введення даних в С використовується виклик функції *scanf( )*. Функція зчитує з консолі значення зазначеного специфікацією формату типу. Зчитування виробляється до першого символу пробілу або символу '\n'. Особливістю цієї функції є використання в якості 2-го та наступних параметрів покажчиків. Тобто обов'язково повинен бути переданий адрес змінної, в яку функція поверне значення-результат введення з клавіатури.

Для введення одного символу зі стандартного потоку введення використовують функцію *getchar( )*, а для введення рядку – *fgets( )*. Приклади:

```
int a = 0;
printf("Enter A: ");
scanf("%d", &a);
getchar();
printf("A = %d\n", a);

double *pb = (double *)malloc(sizeof(double));
printf("Enter B: ");
scanf("%lf", pb);
getchar();
printf("B = %lf\n", *pb);
free(pb);

for(int i=0; i<5; i++)
{
	printf("Enter data[%d] = ", i+1);
	scanf("%d", &data1[i]);
	getchar();
}
```

```
for(int i=0; i<5; i++) printf("data[%d] = %d\n", i+1, data1[i]);

char names[5][15];

for(int i=0; i<5; i++)
{
	printf("Enter name: ");
	scanf("%s", names[i]);
	getchar();
}

for(int i=0; i<5; i++) printf("Name %d -- %s\n", i+1, names[i]);

char str2[15];
char ch;
int i = 0;
printf("Enter string (max length - 15): ");
do
{
	ch = getchar();
	if(ch != '\n') { str2[i] = ch; i++; }
}
while( (ch != '\n')&&(i<15) );
str2[i]='\0';

printf("String: %s\n", str2);

char  buf[512];
printf ("Enter text: ");
if ( fgets(buf, sizeof(buf), stdin) ) printf ("You entered: %s", buf);
```

Виклик *getchar( )* після *scanf( )* бажаний, якщо в подальшому в коді використовується виклик *getchar( )*, оскільки таким чином очищається внутрішній буфер цих функцій і запобігає їх помилкове спрацьовування.

Константи

У стандарті *C90* введена концепція констант, які визначаються користувачем за допомогою ключового слова *const* – значення не можна змінити безпосередньо. Це може бути корисно в декількох аспектах.

1. Багато об'єктів не змінюються після ініціалізації.

2. Використання символічних констант призводить до більш зручного в супроводі коду, ніж застосування літералів безпосередньо в тексті програми.

3. Покажчики часто використовуються тільки для читання, але не для запису.

4. Більшість аргументів функцій читаються, але не перезаписуються.

Приклади:

```
const int model=90;        // model є константою
const int v[]={1,2,3,4};   // всі v[i] є константами
const int x;               // помилка: немає ініціалізатору
```

```
void f( )
{
        model=200;  // помилка: model тільки для читання
        v[2]++;         // помилка: y[2] тільки для читання
}
```

Покажчики та константи

В операціях з покажчиками беруть участь два об'єкти: сам покажчик та об'єкт, на який він посилається.

Використання ключового слова *const* перед оголошенням покажчика робить константою об'єкт, а не покажчик.

Для оголошення самого покажчика як константи, використовується оператор оголошення **const*, а не просто *. Наприклад:

```
void f1(char* p)
{
        char s[]="Gorm";
        const char* pc=s;       // покажчик на константу
        pc[3]='g';              // помилка: pc вказує на константу
        pc=p;                   // вірно: сам покажчик pc не є константним

        char *const cp=s;       // константний покажчик
        cp[3]='a';              // вірно
        cp=p;                   // помилка: cp є константою

        const char *const cpc=s;        // константний покажчик на константу
        cpc[3]='a';                     // помилка: cpc вказує на константу
        cpc=p;                          // помилка: cpc є константою
}
void g(const X* p)
{
        // в цьому блоці не можна змінити *p
}
void h( )
{
        X val;
        …               // val типу X можна змінювати
        g(&val);
        …
}
```

Деякий об'єкт при зверненні до нього через один покажчик може бути константою, а при зверненні через інший – змінної. Ця властивість особливо корисна для аргументів функцій. Оголосивши аргумент-покажчик константою, функція не зможе змінити об'єкт, на який він посилається. Наприклад:

char* strcpy(char* p, const char* q); // не можна змінити *q

Можна привласнити адресу змінної покажчику на константу, бо це нешкідлива операція.

Неможливо привласнити адресу константи будь-якому покажчику, тому що в цьому випадку можна буде змінити значення об'єкту.

Наприклад:

```
void f4()
{
	int a=1;
	const int c=2;
	const int* p1=&c; // правильно
	const int* p2=&a; // правильно
	int* p3=&c;	// помилка:
			// ініціалізація int* значенням типу const int
	*p3=7;		// спроба змінити значення константи c
}
```

Завдання 5.1. Написати функцію обчислення мінімального, максимального, середнього арифметичного та стандартного відхилення для одновимірної вибірки даних з генеральної сукупності. Формула знаходження стандартного відхилення для вибірки з генеральної сукупності:

$$STDDEV = \sqrt{\frac{\sum_{i=1}^{n} (x - \bar{x})^2}{n-1}}$$

Приклад реалізації:

```
// файл vector_stat.cpp
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

void getStatistics(const double * v, int count, double * min, double * max,
                   double * avg, double * stddev);

int main()
{
	double * data = NULL; int count = 0;

	printf("Enter vector size:");  scanf("%d", &count);

	if(count > 1)
	{
		data = (double*)malloc( sizeof(double)*count );
		if(NULL != data)
		{
```

```
                        for(int i=0; i<count; i++)
                        {
                                printf("Enter DATA[%d] = ", i+1);
                                scanf("%lf", &data[i]);
                        }

                        double min, max, avg, stddev;
                        min = max = avg = stddev = 0;
                        getStatistics(data, count, &min, &max, &avg, &stddev);
                        printf("MIN=%.3lf\tMAX=%.3lf\tAVG=%.3lf\tSTDDEV=%lf\n",
                                min, max, avg, stddev);
                        free(data);
                        data = NULL;
                }
        }
        else printf("WARNING!!! count <= 1!!!\n");

        return 0;
}

void getStatistics(const double * v, int count, double * min, double * max,
                        double * avg, double * stddev)
{
        if(count>1)
        {
                *min = *max = *avg = v[0];
                for(int i=1; i<count; i++)
                {
                        if(*min > v[i]) *min = v[i];
                        if(*max < v[i]) *max = v[i];
                        (*avg) += v[i];
                }
                (*avg) /= (double)count;
                for(int i=0; i<count; i++) (*stddev) += pow( (v[i] - *avg) , 2.0);
                *stddev = sqrt( *stddev/(double)(count-1.0) );
        }
}
```

Завдання 5.2. У матриці *matrix* розміром *NxN* кожен елемент розділити на діагональний, що стоїть в тому ж стовпці.

Приклад реалізації:

```
// file ex_1.cpp
#include <stdio.h>
#include <stdlib.h>

float ** getMatrix(float ** matrix, int * count);
void printMatrix(const float ** matrix, int count);
float ** calculateMatrix(float ** matrix, int count);
void destroyMatrix(float ** matrix, int count);

int main()
{
        float ** matrix = NULL;
        int count = 0;
        matrix = getMatrix(matrix, &count);
        if(NULL != matrix)
        {
                printf("Input data:\n");
                printMatrix((const float**)matrix, count);
                matrix = calculateMatrix(matrix, count);
                printf("\nOutput data:\n");
```

```
		printMatrix((const float**)matrix, count);
		destroyMatrix(matrix, count);
	}
	return 0;
}

float ** getMatrix(float ** matrix, int * count)
{
	printf("Enter N-size matrix (matrix NxN):");
	scanf("%d", count);
	matrix=(float**)calloc( *count, sizeof(float*));
	if(NULL != matrix)
	{
		for(int i=0; i<(*count); i++) matrix[i]=(float*)calloc( *count, sizeof(float));
		for(int i=0; i<*count; i++) for(int j=0; j<*count; j++) matrix[i][j] = (i+1)*(j+1);
	}
	return matrix;
}

void printMatrix(const float ** matrix, int count)
{
	if(NULL != matrix) {
		for(int i=0; i<count; i++) {
			for(int j=0; j<count; j++) printf(" %.2e", matrix[i][j]);
			printf("\n");
		}
	}
}

float ** calculateMatrix(float ** matrix, int count)
{
	if(NULL != matrix) {
		for(int j=0; j<count; j++) {
			float v = matrix[j][j];
			for(int i=0; i<count; i++) if(v!=0) matrix[i][j] /= v;
		}
	}
	return matrix;
}

void destroyMatrix(float ** matrix, int count)
{
	for(int i=0; i<count; i++) free(matrix[i]);
	free(matrix);
	matrix = NULL;
}
```

Завдання 5.3. У масиві *mas* кожен третій елемент замінити напівсумою двох попередніх, а той, що стоїть перед ним – напівсумою сусідніх з ним елементів. Додатковий (робочий) масив не використовувати.

Приклад реалізації:

```
// file ex_2.cpp
#include <stdio.h>
#include <stdlib.h>

float * getArray(float * v, int * count);
void printArray(const float * v, int count);
void transformArray(float * v, int count);

int main()
{
	float * mas = NULL;
	int count = 0;
	mas = getArray(mas, &count);
	if(NULL != mas)
```

```
	{
		printf("Input data:\n");
		printArray((const float*)mas, count);
		transformArray(mas, count);
		printf("\nOutput data:\n");
		printArray((const float*)mas, count);
		free(mas);
	}
	return 0;
}

float * getArray(float * v, int * count)
{
	printf("Enter size array:");
	scanf("%d", count);
	v=(float*)calloc( *count, sizeof(float));
	if(NULL != v) for(int i=0; i<*count; i++) v[i] = (i+1);
	return v;
}

void printArray(const float * v, int count)
{
	if(NULL != v) for(int i=0; i<count; i++) printf(" %.2e", v[i]);
	printf("\n");
}

void transformArray(float * v, int count)
{
	if(NULL != v)
	{
		float post_elem = 0; // temporary element!
		if(count>=3) post_elem = v[0];
		for(int i=0; i<count; i++)
		{
			if( ((i+1)%3) == 0 )
			{
				//v[i] = (v[i-1]+v[i-2])/2.0f;
				v[i]   = (v[i-1]+post_elem)/2.0f;
				if((i+3)<count)
				{
					post_elem = v[i+1];
					v[i+1] = (v[i+2]+v[i+3])/2.0f;
				}
			}
		}
		printf("\n");
	}
}
```

Завдання 5.4. У масиві *mas* всі ненульові елементи замінити зворотними за величиною та протилежними за знаком.

Приклад реалізації:

```
// file ex_3.cpp
. . .
void transformArray2(float * v, int count);
. . .
transformArray2(mas, count);
. . .
if(NULL != v) for(int i=0; i<*count; i++) v[i] = i-(*count/2.0);
. . .
void transformArray2(float * v, int count)
{
	if(NULL != v)
	{
```

```
		for(int i=0; i<count; i++) if(0 != v[i]) v[i] = (1.0f / v[i]) * -1.0f;
	}
}
```

Завдання 5.5. Всі парні елементи цілочисельного масиву K помістити в масив L, а непарні – в масив M. Підрахувати кількість тих та інших.

Приклад реалізації:

```
// file ex_4.cpp
. . .
int main()
{
	int * K = NULL; int * L = NULL; int * M = NULL;
	int count_k = 0, count_l = 0, count_m = 0;
	K = getArray(K, &count_k);
	if(NULL != K) {
		printf("Input data:\n");
		printArray((const int*)K, count_k);

		for(int i=0; i<count_k; i++)
		{
			if(K[i]!=0) {
				if((K[i]%2) == 0) count_l++;
				else count_m++;
			}
		}

		L=(int*)calloc(count_l, sizeof(int));
		M=(int*)calloc(count_m, sizeof(int));
		int j = 0; int t = 0;
		for(int i=0; i<count_k; i++)
		{
			if(K[i]!=0) {
				if((K[i]%2) == 0) L[j++] = K[i];
				else M[t++] = K[i];
			}
		}

		printf("\nOutput data 1 (count = %d):\n", count_l);
		printArray((const int*)L, count_l);
		printf("\nOutput data 2 (count = %d):\n", count_m);
		printArray((const int*)M, count_m);

		free(M); free(L); free(K);
	}
	return 0;
}
. . .
```

Завдання 5.6. Написати функції звернення до елементу одновимірного масиву цілих величин, як до елементу матриці, визначеної розмірності, та функції звернення до елементу матриці цілих, як до елементу одновимірного масиву.

Приклад реалізації:

```
int getIndexFromMatrix(int row, int col, int cols) {  return row*cols+col; }

int getElFromArrayAsMatrix(const int * arr, int row, int col, int cols)
{
```

```
	return arr[ getIndexFromMatrix(row, col, cols) ];
}

void getIndexesFromArray(int index, int cols, int *row, int *col)
{
	*row = index / cols;
	*col = index % cols;
}

int getElFromMatrixAsArray(const int ** matrix, int cols, int index)
{
	int r = 0, c = 0;
	getIndexesFromArray(index, cols, &r, &c);
	return matrix[r][c];
}
```

Лекція 6. Елементи мови С++: оператори new та delete. Робота з динамічними масивами

Одномісні оператори *new* та *delete* використовуються для управління *вільною пам'яттю*. Програміст створює об'єкт, використовуючи *new*, і видаляє об'єкт з використанням *delete*.

Оператор *new* може мати такі форми:

new і*м'я_типа*
new і*м'я_типа ініціалізатор*
new (і*м'я_типа*)

У кожному разі отримуємо принаймні, два результату. Спочатку з вільної пам'яті розподіляється обсяг, достатній для того, щоб утримувати іменований тип. Потім повертається базовий адреса об'єкту як значення виразу *new*.

Якщо *new* не виконується, повертається покажчик, що має значення 0.

Третя форма використання *new* застосовується тоді, коли потрібно укласти в дужки ім'я типу для правильного аналізу об'єкта.

Ініціалізатор – список параметрів, укладений в дужки. Ініціалізатор не може використовуватися для ініціалізації масивів.

Оператор *delete* використовується в наступних формах:

delete *вираз*
delete [] *вираз*

В обох формах вираз – зазвичай змінна-покажчик, що використовувалася в попередньому виразі *new*. Друга форма застосовується при поверненні пам'яті, яка розподілялася як тип масиву.

Наприклад:

int * pi = new int; – розміщує об'єкт типу *int* в пам'яті та ініціалізує покажчик *pi* адресою цього об'єкту; об'єкт в такому разі не ініціалізується;

int * pi = new int (1024); – розміщує об'єкт типу *int* в пам'яті та ініціалізує покажчик *pi* адресою цього об'єкта; об'єкт в такому випадку ініціалізується значенням 1024;

int * pia = new int [10]; – динамічно виділяє пам'ять під масив з десяти елементів;

delete pi; delete ps; delete [] pia; – звільнення пам'яті від об'єктів зазначених типів;

```
int* pint;
pint=new int(9); // покажчик на int, int-комірка ініціалізується значенням 9
printf("%d\n", *pint); // звернення до значення за покажчиком
...
// робота з динамічним вектором
int* data = new int[25];
...
for(int i=0;i<25;i++)  data[i]=0;
...
delete [] data;
...
// робота з динамічною матрицею
double ** pmatrix;
pmatrix=new double*[m_size_row];
for(int i=0;i<m_size_row;i++) pmatrix[i]=new double[m_size_col];
...
for(int i=0;i<m_size_row;i++)
     for(int j=0;j<m_size_col;j++)
              pmatrix[i][j]=0;
...
for(int i=0;i<m_size_row;i++)  delete [] pmatrix[i];
delete [] pmatrix;
```

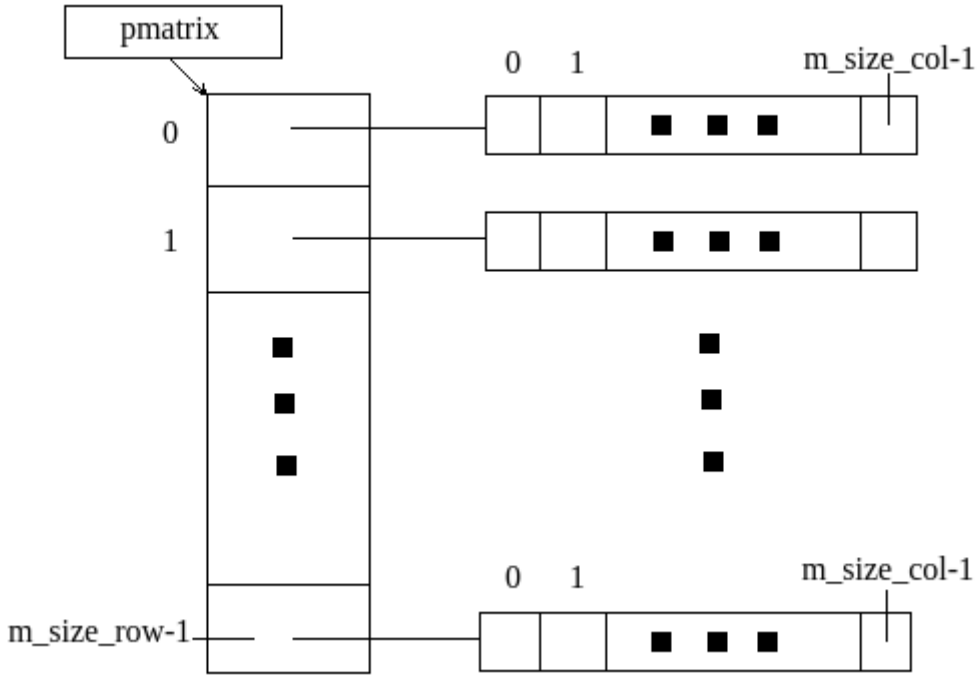


Рис. 6.1. Представлення динамічного двовимірного масиву в пам’яті

Лекція 7. Передача даних програмі через командний рядок та їх обробка. Переведення даних з рядка в число

Для передачі даних програмі через командний рядок необхідно за ім'ям файлу, що виконується, задати аргументи. Аргументи повинні бути відокремлені один від одного пробілами або символами горизонтальної табуляції. Якщо потрібно передати програмі аргумент, що містить в собі прогалини або символи горизонтальної табуляції, слід укласти його в подвійні лапки. Аргументи передаються програмі (функції *main( )*) як символьні рядки.

Наприклад:

c:\> myprog.exe 25 "ab c" 100 ab

У прикладі програмі з ім'ям *myprog* передаються чотири аргументи – символьні рядки: "25", "ab c", "100", "ab".

В процесі компонування програми на мові C/C++ в її склад включається модуль підтримки виконання. Цей модуль отримує управління безпосередньо від ОС при виклику програми, розбирає командний рядок та передає аргументи функції *main( )*.

Приклади заголовку функції *main( )* з параметрами:

```
int main( int argc, char * argv[] )
int main( int argc, char * argv[], char * envp[] )
```

Параметр *argv* функції *main( )* являє собою масив покажчиків, кожен елемент якого вказує на строкове представлення наступного по порядку аргументу. Параметр *argc* визначає загальне число переданих аргументів.

Перший елемент масиву *argv[0]* завжди містить ім'я програми, що виконується. Цей елемент завжди заповнений, тому завжди *argc >= 1*. Останній параметр програми завжди *argv[argc-1]*.

Кількість рядкових параметрів в *envp* можна отримати, просканувавши всі елементи *envp* та порівнюючи результат з NULL.

Приклад:

```
#include <stdio.h>
int main(int argc, char * argv[], char * envp[])
{
   for(int i=0; i<argc; i++) printf("ARGV[%d] = %s\n", i, argv[i]);
   puts("================================");
   for(int i=0; envp[i]!=NULL; i++) printf("ENVP[%d] = %s\n", i, envp[i]);

   return 0;
}
```

Для переведення рядка в число, використовується ряд функцій перетворення переданих значень програмі через командний рядок. Наприклад (файл stdlib.h):

int atoi (const char * nptr); – з рядка в ціле
long atol (const char * nptr); – з рядка в довге ціле
double atof (const char * nptr); – з рядка в дійсне

Наведені функції не проводять перевірку на помилки. Наприклад, якщо передати не число, представлене рядком, а іншу строкову інформацію, то функції повернуть нуль. Наприклад:

```
#include <stdlib.h>
. . .
int a = 0;
. . .
a = atoi( "10" );    // a = 10
a = atoi( "abc" );  // a = 0
. . .
```

Для встановлення факту помилки конверсії використовують, наприклад, функції *strtol( ), strtof( ), strtod( )* та ін. Наприклад:

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <limits.h>
#include <math.h>

#define FALSE     0
#define TRUE      1

int StrToIntBase10(const char * str, long * val);
int StrToDouble(const char * str, double * val);

int main(int argc, char *argv[])
{
      int a = -1;
      double b = -1;

      if( StrToIntBase10("25", (long *)&a) == FALSE )
            printf("Integer Conversion Error!\n");

      printf("A = %d\n", a);

      if( StrToDouble("-275.3579e-3", &b) == FALSE )
            printf("Double Conversion Error!\n");
```

```
	printf("B = %lf\n", b);

	return 0;
}


int StrToIntBase10(const char * str, long * val)
{
	if((NULL != val)&&(NULL != str))
	{
		errno = 0;
		char * endptr;
		*val = 0;

		*val = strtol(str, &endptr, 10);

	  if ((errno == ERANGE && (*val == LONG_MAX || *val == LONG_MIN))
		|| (errno != 0 && *val == 0))
		{
			*val = 0;
			return FALSE;
		}

		if (endptr == str)
		{
			*val = 0;
			return FALSE;
		}

		return TRUE;
	}
	else return FALSE;
}

int StrToDouble(const char * str, double * val)
{
	if((NULL != val)&&(NULL != str))
	{
		errno = 0;
		*val = 0;
		char * endptr;

		*val = strtod(str, &endptr);

	  if ( (errno == ERANGE && (*val == -HUGE_VAL || *val == HUGE_VAL))
		|| (errno != 0 && *val == 0))
		{
			*val = 0;
			return FALSE;
		}

		if (endptr == str)
		{
			*val = 0;
```

```
                return FALSE;
            }

            return TRUE;
        }
        else return FALSE;
}
```

Завдання. Реалізувати функції та основну програму, яка обчислює середнє арифметичне масиву дійсних чисел, переданих програмі в якості аргументів командного рядку. Обчислення реалізувати на базі функції:

double getAvg(const double *, int count)

Ігнорувати помилки конверсії. Сповіщати користувача про інші помилки програми за допомогою функції:

int getError(const char * prog_name, int errID)

Приклад реалізації:

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

#define ERR_PARAM   1
#define ERR_MEM     2

int getError(const char * prog_name, int errID);
double getAvg(const double * v, int count);

int main(int argc, char * argv[])
{
       if(argc==1) return getError(argv[0], ERR_PARAM);

       double *pdata = (double *)malloc(sizeof(double)*(argc-1));
       if(NULL != pdata)
       {
              for(int i=1; i<argc; i++) pdata[i-1] = atof(argv[i]);
              printf("Average value = %lf\n", getAvg(pdata, argc-1));
              free(pdata); pdata = NULL;
       }
       else return getError(NULL, ERR_MEM);

       return 0;
}

int getError(const char * prog_name, int errID)
{
       switch(errID)
       {
              case ERR_PARAM:
                     puts("Error parameters!");
                     printf("Example:\n%s 1 2 3 4 5\n", prog_name);
                     break;
```

```
		case ERR_MEM:
			puts("Memory allocation error!");
			break;
	}

	return errID;
}

double getAvg(const double * v, int count)
{
	double sum = 0;
	 if(count>0)
	{
		for(int i=0; i<count; i++) sum += v[i];
		return sum/(double)count;
	}
	else return 0;
}
```

Лекція 8. Покажчик на функцію. Оголошення типів даних користувача. Перерахування

Покажчик на функцію

З функцією можна робити тільки дві операції: викликати її та отримати її адресу. Покажчик, отриманий взяттям адреси функції, можна потім використовувати для виклику функції. Наприклад:

```
#include <stdio.h>
void (*efct) (char*);     // покажчик на функцію
void error(char* s);
int main( ) {
        efct=&error;        // efct вказує на функцію error
        efct("ошибка");   // виклик error через efct
        return 0;
}
void error(char* s) { puts(s); }
```

Компілятор розпізнає, що *efct* є покажчиком та викличе функцію, на яку він вказує. Тобто, розіменування покажчика на функцію за допомогою оператора * є необов'язковим. Аналогічно, необов'язково користуватися & для отримання адреси функції:

```
int main( ) {
  void (*f1) (char*) = &error; // правильно
  void (*f2) (char*) = error; //правильно і означає то ж саме, що й &error
  f1("Test");          // правильно
  (*f1) ("Test2");   // правильно
  return 0;
}
```

Конструкція typedef

Оголошення, що починається з ключового слова *typedef*, вводить нове ім'я для типу, а не для змінної даного типу. Наприклад:

```
typedef char* PChar;
PChar p1, p2;                // p1 та p2 фактично мають тип char*
char* p3=p1;                 // правильно
```

Метою такого оголошення часто є призначення короткого синоніма для типу, що часто використовується. Наприклад, при частому застосуванні *unsigned char* можна ввести синонім *uchar* або *byte* що є еквівалентом:

typedef unsigned char uchar;
typedef unsigned char byte;

Інше використання *typedef* – звести в одне місце всі безпосередні посилання на якийсь тип. Наприклад:

typedef int int32;
typedef short int16;

Таким чином, використовуючи *int32* всюди, де можуть бути потрібні великі числа, можна перенести додаток на машину з *sizeof(int) == 2*, просто замінивши єдину строчку коду з *int*:

typedef long int32;

Імена, що вводяться *typedef*, є синонімами, а не новими типами.
Часто прийнято визначати імена для типів покажчиків на функції, щоб уникнути постійного вживання неочевидного синтаксису їх оголошень.
Наприклад:

```
typedef int (*PF) (int[ ], int);    // PF – новий тип
typedef void (*funcptr)( );         // funcptr – новий тип
funcptr table[10];                  // еквівалентно "int (*table[10])();"
```

Приклад використання покажчика на функцію (*f* – покажчик на функцію з одним аргументом дійсного типу, яка повертає дійсне значення; тип таких функцій оголошений як MATH_FUNC):

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

typedef double(*MATH_FUNC)(double);
double func( const double * p, const int size, MATH_FUNC f );

int main( )  {
     double vector[]={0.3, 0.7, 0.9};
     int size = 3;
     printf("Result = %lf\n", func(vector, size, sin) );
```

```
        printf("Result = %lf\n", func(vector, size, cos) );
        return 0;
    }

    double func(const double * p, const int size, MATH_FUNC f)
    {
        double s = 0;
        for(int i=0; i<size; i++) s += f(p[i]);
        return s;
    }
```

Перерахування

Тип *enum* (перерахування) задає набір значень, який визначається користувачем. Після свого визначення, перерахування використовується майже так само, як і цілі типи.

В якості елементів перерахування можна визначити іменовані цілі константи. Наприклад:

```
#include <stdio.h>
typedef enum {READ, WRITE, READWRITE } MODE;
void doProcess(void* f, MODE fmode);
int main( )
{
    doProcess(NULL, READWRITE);
    doProcess(NULL, WRITE);
    doProcess(NULL, READ);
    return 0;
}

void doProcess(void* f, MODE fmode) {
    switch(fmode)  {
        case READ:      puts("Read mode");  break;
        case WRITE:     puts("Write mode"); break;
        case READWRITE: puts("R/W mode");   break;
    }
}
```

Тип MODE є перерахуванням та визначає три цілі константи – елементи перерахування, яким присвоюються цілі значення – 0, 1, 2.

Перерахування – зручний спосіб асоціювати значення констант з символічними іменами з тою перевагою перед #define, що значення можуть генеруватися автоматично.

Приклад перерахування з явною вказівкою значень констант:

```
#include <stdio.h>
int main( ) {
      enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
                     NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

      printf("%cThis%cis%ctest", BELL, TAB, NEWLINE);
      return 0;
}
```

Лекція 9. Структури та об'єднання. Підтримка кирилиці в консольних програмах

Структури

Якщо елементи масивів є однотиповими, то структури можуть включати в себе елементи різних типів даних.

Оголошення структури починається з ключового слова *struct*, за яким слідує ім'я структури. Далі в фігурних дужках оголошуються змінні певних типів, які є полями структури, її елементами.

Наприклад, нехай необхідно створити структуру, яка буде описувати інформацію про співробітника (employee) деякого підрозділу. Така інформація повинна містити в собі, як мінімум, деякий ідентифікатор співробітника, ідентифікатор підрозділу, П.І.Б. співробітника. Виходячи з цього опису можна зробити таке оголошення структури:

```
struct Employee {
        int ID;
        int subdivisionID;
        wchar_t name[15];
        wchar_t surname[20];
        wchar_t patronymic[20];
};
```

Досить часто в програмах на C/C++, структури оголошують як новий тип даних користувача. Наприклад, оголошення двох нових типів даних EMPLOYEE та PEMPLOYEE (для роботи зі структурою через покажчик):

```
typedef struct _tagEmployee {
        int ID;
        int subdivisionID;
        wchar_t name[15];
        wchar_t surname[20];
        wchar_t patronymic[20];
}EMPLOYEE, *PEMPLOYEE;
. . .
EMPLOYEE e1, e2, e3;  // оголошення змінних типу структури EMPLOYEE
PEMPLOYEE pe;
```

Звернення до полів структур в разі, коли змінна не є покажчиком, проводиться через символ «**.**», а в разі змінної-покажчика – через зв'язку символів «->». Наприклад:

```
// Файл emp.cpp в кодуванні UTF-8
// з записаною сигнатурою BOM (0xEF 0xBB 0xBF) на початку файлу.
// Сигнатура додається при обранні, наприклад, в редакторах Notepad++, Geany.
// У випадку збірки в Code::Blocks цього можна не виконувати (див. далі).
#include <stdio.h>
#include <string.h>
#include <wchar.h>
#include <locale.h>

#ifdef __WINDOWS__
    #include <windows.h>
#endif

typedef struct _tagEmployee {
   int ID;
   int subdivisionID;
   wchar_t name[15];
   wchar_t surname[20];
   wchar_t patronymic[20];
}EMPLOYEE, *PEMPLOYEE;

int main()
{
   // для організації підтримки виведення на консоль
   // кирилічних символів кодових таблиць
   #ifdef __WINDOWS__
      setlocale(LC_ALL, "Ukrainian");
   #else
      setlocale(LC_ALL, "uk_UA.utf8");
   #endif

   EMPLOYEE e;
   PEMPLOYEE pe;

   e.ID = 1;
   e.subdivisionID = 2;
   wcscpy(e.name, L"Тарас");
   wcscpy(e.surname, L"Григорович");
   wcscpy(e.patronymic, L"Шевченко");

   pe = &e;

   wprintf(L"%d\t%d\t%ls\t%ls\t%ls\n",
           pe->ID, pe->subdivisionID, pe->name, pe->surname, pe->patronymic);

   return 0;
}
```

Перевірка присутності BOM:

MS Windows 10/11: *powershell fhx emp.cpp | more*

GNU/Linux: *hexdump -C ./emp.cpp | head*

Збірка з командного рядку в MS Windows (потрібні додаткові налаштування системної змінної *PATH* для вказівки розташування компіляторів):

GNU GCC MinGW: `g++ -D__WINDOWS__ -o emp emp.cpp`

MS C/C++ Compiler: `cl /D__WINDOWS__ emp.cpp`

Збірка з командного рядку в GNU/Linux:

GNU GCC: `g++ -o ./emp ./emp.cpp`

Примітка: для правильної компіляції описаного прикладу з середовища Code::Blocks в MS Windows 10/11, необхідно вибрати пункт меню *Project \ Build options*. У діалоговому вікні перейти до закладки *Compiler settings \ #defines* та додати визначення макроідентіфікатору:

```
__WINDOWS__
```

Перейти до закладки *Compiler settings \ Other options* та додати новий рядок опцій:

-finput-charset=windows-1251

Після цього виконувати збірку проекту та виконання коду.

Приклад визначення (ініціалізації) полів структури при оголошенні:

EMPLOYEE e = {2, 3, L"Тарас", L"Григорович", L"Шевченко"};

Покажчик на ім'я типу структури можна використовувати негайно після його появи, а не обов'язково після завершення всього оголошення. Наприклад:

```
struct Link {
	int data;
	Link* prev;
	Link* next;
};
```

Такі описи зручні у випадках побудови складних динамічних структур даних, наприклад, динамічних списків, дерев і т.п.

До повного завершення оголошення структури забороняється використовувати її ім'я для оголошення інших об'єктів. Наприклад:

```
struct No_A {
	No_A member; // помилка!
};
```

Подібний запис є помилкою, тому що компілятор не може визначити розмір No_A. Для того щоб два (або більше) об'єкта типу структури посилалися один на одного, можна спочатку оголосити тільки ім'я типу:

```
struct List;
struct Link {
	Link* prev;
	Link* next;
	List* member_of;
};
struct List {
	Link* head;
};
```

При виборі імен структур слід уникати їх збігів з іншими елементами програми, наприклад, іменами функцій.

Завдання 9.1. Написати функції введення/виведення float-значень координат вершин в тривимірному просторі. Вершина описана деякою структурою *XYZ*. Протестувати функції в основній програмі.

```
//coords.cpp
#include <stdio.h>

typedef struct _tagXYZ
{
	float x;
	float y;
	float z;
}XYZ, *PXYZ;

void inputVertexFromConsole(XYZ *pv);
void printVertex(const XYZ v);

int main()
{
	XYZ v;
	inputVertexFromConsole(&v);
	printVertex(v);
	return 0;
}

void inputVertexFromConsole(XYZ *pv)
{
	printf("Enter X:"); scanf("%f", &(pv->x));
	printf("Enter Y:"); scanf("%f", &(pv->y));
	printf("Enter Z:"); scanf("%f", &(pv->z));
}

void printVertex(const XYZ v)
{
	printf("VERTEX Coords: X = %f\tY = %f\tZ = %f\n", v.x, v.y, v.z);
}
```

Завдання 9.2. На основі розробленої вище програми, організувати обробку безлічі вершин, кількість яких на момент розробки програми невідомо і має бути визначено користувачем.

```
. . .
#include <stdlib.h>
. . .
int main()
{
	XYZ* v = NULL;
	int count = 0;
	printf("Enter count vertexes: "); scanf("%d", &count);
	if((v = (XYZ*)malloc(count*sizeof(XYZ)))!=NULL)
	{
		for(int i=0; i<count; i++) { inputVertexFromConsole(&v[i]); puts(""); }
		for(int i=0; i<count; i++) printVertex(v[i]);
		free(v);
	}
	return 0;
}
. . .
```

Об'єднання

Об'єднання – це спеціальна структура в пам'яті, яка може містити об'єкти різних типів і розмірів таким чином, що вони зберігаються в пам'яті за однією й тією ж адресою. Вимоги до розміру і вирівнювання значень полів в пам'яті покладаються на компілятор. За допомогою об'єднань можна працювати з даними різних типів в межах однієї ділянки пам'яті, не привносячи в програму елементи низькорівневого, машинозалежного програмування.

Розмір пам'яті, що відводиться під об'єднання, як правило, відповідає розміру пам'яті, що відводиться для найбільшого типу в об’єднанні.

Для оголошення об'єднання використовується ключове слово *union*. Правила звернення до полів об'єднання такі ж, як і до полів структур. Наприклад:

```
#include <stdio.h>

typedef unsigned char byte;

typedef union _tagMemoryInteger32
{
	int v;
	byte mem[4];
}MEMORY_INT32;

int main()
{
	MEMORY_INT32 a;
	a.v = 32767;
	printf("DEC: %d\n", a.v);
	printf("HEX: %X\n", a.v);
	printf("MEMORY DUMP:");
	for(int i=0; i<4; i++) printf(" %02X", a.mem[i]);
	puts("");
	return 0;
}
```

Приклад виводу:

```
DEC: 32767
HEX: 7FFF
MEMORY DUMP: FF 7F 00 00
```

Як правило типи полів в об'єднанні вибирають таким чином, що б їх подальша обробка мала сенс!

Над об'єднанням можна виконувати ті ж операції, що і над структурами: привласнювати або копіювати як єдине ціле, брати адресу та звертатися до окремих елементів.

Змінні об'єднання можна ініціалізувати тільки даними того типу, який має його перший елемент. Наприклад:

```
MEMORY_INT32 a = { 12 };
```

Лекція 10. Робота з файлами на мові програмування С

Введення/виведення на мові С

Функції введення/виведення (input/output, I/O) в стандартній бібліотеці С дозволяють читати дані з файлів або отримувати їх з пристроїв введення (наприклад, з клавіатури), записувати дані в файли або виводити їх на різні пристрої (наприклад, на принтер, пристрій друку). Функції I/O діляться на три класи.

1. Функції I/O верхнього рівня (з використанням поняття "потік").
2. Функції I/O для консольного терміналу шляхом безпосереднього звернення до нього.
3. Функції I/O нижнього рівня (з використанням поняття "дескриптор" (описувач).

Функції I/O верхнього рівня забезпечують буферизацію при роботі з файлами. Це означає, що коли проводиться читання даних з файлу або запис в файл, обмін даними здійснюється не між програмою і зазначеним файлом, а між програмою і проміжним буфером, розташованим в ОЗП (оперативному запам'ятовувальному пристрої).

Для користувача файл, відкритий на верхньому рівні представляється як послідовність байт, яка зчитується або записується. Щоб відобразити цю особливість організації I/O, запропоновано поняття "потік" (stream). Коли файл відкривається, з ним пов'язується потік. Дані, що виводяться, записуються в потік і зчитуються з потоку.

Коли потік відкривається для I/O, він пов'язується зі структурою типу FILE (ім'я типу FILE визначається за допомогою конструкції *typedef* в файлі stdio.h).

Функції I/O верхнього рівня дають можливість для буферизованого форматованого та неформатованого I/O.

Функції I/O низького рівня не виконують буферизацію та форматування даних. Вони дозволяють безпосередньо користуватися засобами I/O ОС. При низькорівневому відкритті файлу з ним пов'язується дескриптор (*handle*). Дескриптор є цілим значенням, що характеризує розміщення інформації про відкритий файл у внутрішніх таблицях системи. Дескриптор використовується при подальших операціях з файлом.

Функції I/O нижнього рівня зі стандартної бібліотеки доцільно використовувати при розробці своєї власної підсистеми I/O.

Прототипи всіх функцій I/O верхнього рівня (їх оголошення) містяться у файлі *stdio.h*. У цьому файлі визначені також наступні константи:

EOF – код, що повертається як ознака кінця файлу;

BUFSIZ – визначає розмір буфера потоку в байтах.

Стандартні потоки I/O

Коли програма починає виконуватися, автоматично відкриваються три стандартні потоки:

– стандартний потік введення (`stdin`);

– стандартний потік виведення (`stdout`);

– стандартний потік виведення повідомлень про помилки (`stderr`);

За замовчуванням *stdin, stdout* та *stderr* пов'язуються з консольним терміналом.

У файлі *stdio.h* визначаються відповідні покажчики на тип структури *FILE*, наприклад: *FILE * stdin*.

Примітка: в старих ОС типу MS DOS бібліотека С підтримувала ще два потоки: стандартний послідовний порт (`stdaux`) та стандартний пристрій друку на паралельному порту (`stdprn`). В ОС MS Windows, а також в Unix-сумісних ОС подібні потоки за замовчанням в бібліотеці С відсутні.

Функції для роботи з файловими потоками

```
FILE *fopen(const char *pathname, const char *mode);
```

Функція *fopen( )* відкриває файл, ім'я якого задається аргументом *pathname* та пов'язує з ним потік (для виконання високорівневого I/O).

Аргумент *mode* вказує на рядок символів, що визначають спосіб доступу до файлу, і може містити такі значення:

"r" – відкрити для читання (файл повинен існувати);

"w" – відкрити порожній файл для запису (якщо файл існує, то його вміст втрачається);

"a" – відкрити для запису в кінець файлу (додавання);

"r+" – відкрити для читання і запису (файл повинен існувати);

"w+" – відкрити порожній файл для читання і запису (якщо файл існує, то його вміст втрачається);

"a+" – відкрити для читання та додавання (файл створюється, якщо він не існує).

Також в рядку *mode* може бути присутнім один із символів *'t'* (текстовий режим) або *'b'* (бінарний (двійковий)). Наприклад: *"wt"*. За замовчуванням встановлений текстовий режим.

```
int fclose(FILE *stream);
int fcloseall(void);
```

Функції *fclose( )* та *fcloseall( )* закривають потік або потоки, пов'язані з відкритими за допомогою *fopen( )* файлами. Функція *fcloseall( )* закриває всі потоки, за винятком стандартних.

Приклад:

```
#include<stdio.h>
int main( )
{
     FILE * fin = NULL;
     if( ( fin = fopen( "test.txt", "r" ) ) != NULL )
     {
          // виконати дії з файлом test.txt через fin
          // . . .
          fclose( fin );
     }
     return 0;
}
```

Функція *feof( )* визначає досягнутий кінець вказаного потоку.

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

Якщо кінець потоку досягнуто, то буде повернуто код EOF. Якщо кінець файлу недосягнутий – то повертається нульове значення.

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
int getchar(void);
int putchar(int c);
```

Функція *fgetc( )* читає один символ з вхідного потоку stream з поточної позиції та переміщує (внутрішній) покажчик файлу на наступний символ. Виклик функції *getchar( )* відповідає виклику *fgetc(stdin)*. Функції повертають код введеного символу або EOF.

Функція *fputc( )* записує одиночний символ *c* в потік stream в поточну позицію. Виклик *putchar(c)* відповідає виклику *fputc(c, stdout)*. Значення, що повертається – код символу, що записується або EOF.

Приклад реалізації команди командного рядку в терміналі MS Windows "copy" для текстового режиму:

```
#include<stdio.h>
int main(int argc, char * argv[])
{
    FILE * fin = NULL;
    FILE * fout = NULL;

    // Перевіряємо кількість аргументів програми
    if( argc == 3 )
    if( ( fin = fopen( argv[1], "r") ) != NULL )     // відкриваємо вхідний потік
    {
      if( ( fout = fopen( argv[2], "w") ) != NULL ) // відкриваємо потік виведення
      {
           while( !feof( fin ) )  // поки не кінець потоку введення, виконуємо обробку
           {
              int ch = fgetc( fin );  // зчитуємо з потоку символ
              // та якщо це ні EOF, то записуємо його в потік виведення
              if(ch != EOF) fputc( ch, fout );
           }
           // закриваємо обидва потоки, які раніше пов'язані з файлами
           fclose( fout );
           fclose( fin );
      }
      else fclose( fin ); // закриваємо перший потік при невдалому відкриті іншого потоку
    }

    // завершуємо програму
    return 0;
}
```

Функція *fgets( )* читає рядок з вхідного потоку stream та розміщує його в рядок, адреса якого задається значенням параметра *s*.

```
char *fgets(char *s, int size, FILE *stream);
int fputs(const char *s, FILE *stream);
```

Символи читаються з потоку до тих пір, поки не буде прочитаний символ нового рядку ('\n'), який включається в рядок, або поки не настане кінець потоку, або поки не буде прочитано (*size-1*) символів. Результат поміщається в *s* і закінчується нульовим символом ('\0'). Якщо *size* дорівнює 1, то формується порожній рядок. Значення, що повертається – адреса рядка або NULL.

Функція *fputs( )* копіює рядок *s* в потік stream, з поточної позиції. Завершальний нульовий символ ('\0') не буде копіюватися. Значення, що повертається – останній записаний символ, 0 – якщо рядок *s* порожній або значення *EOF*, якщо сталася помилка.

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

Функція *fseek( )* переміщує покажчик файлу (внутрішній покажчик), пов'язаного з потоком stream, на нове місце в файлі, яке обчислюється за зміщенням *offset* і вказівкою напрямку відліку *whence*. Наступна операція I/O із зазначеним потоком stream буде виконана, починаючи з тієї позиції, на яку зроблене переміщення. У потоці, відкритому для зміни, наступною операцією може бути читання або запис. Аргумент *whence* повинен бути однією з наступних констант, визначених в *stdio.h*:

SEEK_SET – початок файлу;

SEEK_CUR – поточна позиція файлу;

SEEK_END – кінець файлу.

Для текстового режиму роботи з файлом значення аргументу *offset* має бути отримано за допомогою функції *ftell( )* або дорівнювати нулю. Функція повертає 0, якщо покажчик файлу був успішно переміщений, нульове значення в разі помилки або невизначеності значення для пристроїв термінал або принтер.

Функція *ftell( )* дозволяє отримати поточну позицію покажчика (внутрішнього) файлу, пов'язаного з потоком stream. Позиція задається як зміщення відносно початку файлу. Значення, що повертається (-1L) сигналізує про помилку. Для терміналу або принтера повертається значення, яке не визначене.

Функція *rewind( )* переміщує покажчик (внутрішній) файлу, пов'язаного з потоком stream, на початок файлу. Виклик функції:

```
rewind(stream);
```

відповідає виклику функції:

```
fseek(stream, 0L, SEEK_SET);
```

за винятком того, що функція *rewind( )* очищує ознаку кінця файлу і ознаку помилки, а функція *fseek( )* цього не робить.

Приклад програми визначення розміру дискового файлу:

```
#include<stdio.h>
int main(int argc, char * argv[] )
{
    FILE * f = NULL;
    if(argc == 2)
      if( ( f=fopen( argv[1] , "rb" ) ) != NULL )
      {
          fseek( f , 0L , SEEK_END );
          long FileSize = ftell ( f );
          fclose( f );
          printf( "Розмір файлу %s складає %ld байт\n", argv[1] , FileSize );
      }
    return 0;
}
```

Функції *fscanf( )* та *fprintf( )* є практично повною аналогією функцій форматованого I/O: *scanf( )* та *printf( )* й спрямовані на роботу з файловими потоками.

```
int fscanf(FILE *stream, const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
FILE *freopen(const char *pathname, const char *mode, FILE *stream);
```

Функція *freopen( )* закриває файл, пов'язаний в поточний момент з потоком stream, та перепризначає потік stream файлу, який визначений у *pathname*. Новий файл, пов'язаний з потоком stream, відкривається з певним типом доступу *mode*. Функція повертає покажчик потоку для нового відкритого файлу або NULL в разі помилки, при цьому спочатку відкритий файл закривається.

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

Функція *fflush( )* скидає вміст буфера, пов'язаного з потоком stream в файл, пов'язаний з цим потоком. Значення, що повертаються: 0 – якщо буфер успішно скинуто або EOF у випадку помилки.

Виклик функції *fflush(NULL)* записує вміст всіх буферів, пов'язаних з відкритими потоками. Всі потоки залишаються відкритими після виконання цієї функції. Функція обнуляє також всі буфера, пов'язані з файлами, відкритими на читання.

У загальному випадку буфер потоку автоматично скидається, коли він заповнюється або коли потік закривається, або ж коли програма завершилася нормально без закриття потоку.

У разі бінарного I/O (в функції *fopen( )* необхідно вказати відповідний режим, наприклад, "rb", "wb" або інший) використовують функції *fread( )* та *fwrite( )*, які мають такі заголовки:

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

Першими параметрами в ці функції передаються покажчики на буфер пам'яті, в який необхідно завантажити байтові дані з потоку або вміст якого необхідно записати в потік. Другими параметрами вказується обсяг пам'яті під одиничний елемент буфера, а кількість таких елементів передається в третьому параметрі. Наприклад:

```
#include <stdio.h>
. . .
int iOutBuff= 25;
int iInBuff = 0;
. . .
```

```
fwrite( &iOutBuff, sizeof( int ), 1, fout );
. . .
fread( &iInBuff, sizeof( int ), 1, fin );
. . .
```

Функції *fread( )* та *fwrite( )* повертають кількість завантажених або записаних елементів. Якщо кількість відрізняється від запланованої, то сталася помилка або досягнуто кінець файлу (в разі читання).

Лекція 11. Рядки

Рядкові константи на мові С – це по суті масиви символів. У внутрішньому представленні рядок закінчуються нульовим символом '\0' – ознакою кінця рядку. Тобто пам'ять, що виділяється під рядок на один символ довше, ніж задано в рядку.

Приклад різних визначень рядків:

```
const char * pstr = "String1";
char str [] = "String2";
```

У першому випадку копіювання рядка не виконується. У мові С немає операцій для маніпулювання рядками як єдиним цілим. Замість цього застосовуються операції з покажчиками.

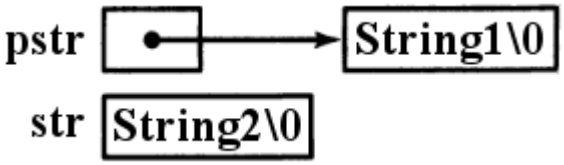


Рис. 11.1. Представлення рядків

Окремі символи в *str* можна змінювати, але змінна *str* завжди буде вказувати на одну й ту ж ділянку пам'яті. *pstr* – константний покажчик, в разі якого компілятор не дозволить змінити вміст рядку.

Приклад реалізації програми найпростішої обробки рядків:

```
#include <stdio.h>
#define MAX_SIZE 512
int GetStringLength(const char *pstr);
void GetString(char * pstr, int max_size);
int main()
{
	char str[MAX_SIZE]="";
	printf("Enter string: ");
	GetString(str, MAX_SIZE);
	printf("String - \"%s\", Length - %d\n", str, GetStringLength(str));

	return 0;
}

void GetString(char * pstr, int max_size)
{
	fgets(pstr, max_size, stdin);
	pstr[GetStringLength((const char*)pstr)-1] = '\0';
}
```

```
int GetStringLength(const char *pstr)
{
	int count = 0;
	for(; pstr[count] != '\0'; count++) ;
	return count;
}
```

Приклади реалізації функцій копіювання символів з одного рядку в інший:

```
char * StringCopy1(char *dst, const char *src)
{
	int i = 0; while(src[i]!='\0') { dst[i]=src[i]; i++; }
	dst[i]='\0';
	return dst;
}

char * StringCopy2(char *dst, const char *src)
{
	while(*dst++ = *src++) ;
	return dst;
}
```

У разі функції *StringCopy2( )* маємо скорочений варіант: пріоритет виконання операцій наступний:

1. *dst = *src – копіюємо символ
2. src ++, dst ++ – змінюємо адресу
3. Якщо результат в *dst НЕ дорівнює 0, то переходимо до кроку 1, інакше завершуємо цикл.

Для роботи з рядками в С використовуються функції, оголошення яких представлені в файлі *string.h*. Найбільш часто використовуваними є наступні:

– strlen(s) – повертає довжину рядка;

– strcpy(dst, src) – копіює з src в dst: dst = src;

– strcmp(s1, s2) – порівняння рядків (0: s1 == s2; <0: s1<s2; >0: s1>s2);

– strcasecmp(s1, s2) – порівняння рядків з ігноруванням регістру;

– strcat(dst, src) – об'єднує (конкатенує) рядки: dst = dst + src;

– strchr(s, ch), strrchr(s, ch) – повертають підрядок з знайденого першого символу (*strchr*) з початку рядка і з першого від кінця рядка (*strrchr*);

– strtok_r(s, delim, &res) – повертає частину рядка, яка розділена роздільниками. Наприклад:

```
	strcpy(str, "123 456 789");
	char *res;
	while(strlen(str)>0)
	{
		printf("%s\n", strtok_r(str, " ", &res));
		strcpy(str, res);
	}
```

Завдання 1. Написати функцію, яка виконує реверс вмісту рядка.
Приклад реалізації:

```
char * reversString(char * dst, const char* src)
{
	int start = strlen(src)-1;
	for(int i=start; i>=0; i--) dst[start-i] = src[i];
	dst[start+1] = '\0';
	return dst;
}
```

Завдання 2. Написати функцію, яка визначає входження деякого підрядку в рядок та повертає 1, якщо підрядок там є. Якщо підрядка в рядку не існує, то повертає значення 0.
Приклад реалізації:

```
int findSubStr(const char * str, const char * substr)
{
	int res = 0;

	int len1 = strlen(str);
	int len2 = strlen(substr);

	char tmpstr[128] = ""; // критична кількість символів в підрядку

	for(int j=0; j<len1; j++)
	{
		if((len1-j)<len2) break;
		int i=0;
		while((i<len2)&&((j+i)<=len1))
		{
			tmpstr[i] = str[j+i];
			i++;
		}
		tmpstr[i]='\0';
		if( strcmp(substr, tmpstr)==0 ) { res = 1; break; }
	}

	return res;
}
```

На мові програмування C++ для роботи з рядками використовують спеціальний клас (тип даних) *string*, який оголошений в header-файлі *string* в стандартній бібліотеці C++. Для повернення С-рядку з об’єкту *string* є спеціальна функція *c_str( )*:

```
#include <string>
using namespace std;
. . .
string s( "test" );
. . .
printf( "s = %s\n", s.c_str( ) );
```

Завдання та приклади рішень

Завдання 1. Напишіть на мові С програму, яка здійснює заповнення локального одновимірного масиву з десяти елементів типу *float*, десятьма значеннями, які передані через параметр *argv* функції *main( )*, а після цього виводить значення елементів масиву на екран.

Приклад реалізації:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int main(int argc, char * argv[])
{
	float data[MAX] = {0};

	if(argc!=(MAX+1)) {
		printf("Number of items ERROR!\n");
		printf("Example: ex1 1 2 3 4 5 6 7 8 9 10\n");
		return 1;
	}

	for(int i=0; i<MAX; i++)
	{
		data[i] = atof(argv[i+1]);
		printf("DATA[%2d] = %f\n", i+1, data[i]);
	}
	return 0;
}
```

Завдання 2. Напишіть на мові С програму, яка виробляє заповнення одновимірного масиву з десяти елементів типу *float*, десятьма значеннями, переданими основній програмі через змінну *argv*, а після цього виводить значення елементів масиву в текстовий файл, ім'я якого передано в якості першого параметра програми в командному рядку.

Приклад реалізації:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int main(int argc, char * argv[])
{
	FILE * fout = NULL;
	float data[MAX] = {0};

	if(argc!=(MAX+2)) {
		printf("Number of items ERROR!\n");
		printf("Example: ex2 output.txt 1 2 3 4 5 6 7 8 9 10\n");
		return 1;
	}
```

```
	if((fout = fopen(argv[1], "w")) != NULL)
	{
		for(int i=0; i<MAX; i++)
		{
			data[i] = atof(argv[i+2]);
			fprintf(fout, "%f ", data[i]);
		}
		fprintf(fout, "\n");
		fclose(fout);
	}
	return 0;
}
```

Завдання 3. Напишіть на мові С програму, яка виводить на екран середнє арифметичне одновимірного масиву дійсних чисел, переданих програмі через командний рядок. Кількість елементів може бути будь-якою.

Приклад реалізації:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char * argv[])
{
	if(argc == 1) {
		printf("Number of items ERROR!\n");
		printf("Example: ex3 1 2 3 4 5 ...\n");
		return 1;
	}
	double sum = 0.;
	for(int i=1; i<argc; i++) sum += atof(argv[i]);
	printf("Average = %lf\n", sum / (argc-1));
	return 0;
}
```

Завдання 4. Реалізуйте на мові С функцію int GetStrLen(const char *) та програму, яка тестує її можливості. Функція повертає кількість символів в переданому рядку. Заборонено використовувати бібліотечні функції обробки рядків. Максимальна кількість символів у рядку не може перевищувати 512.

Приклад реалізації:

```
#include <stdio.h>
#define MAX_STR_LENGTH 512
int GetStrLen(const char *);
int main()
{
	char strTest[MAX_STR_LENGTH] = "";
	printf("Enter string:");
	if( fgets(strTest, MAX_STR_LENGTH, stdin) != NULL )
	{
		printf("String length: %d\n", GetStrLen(strTest));
	}
	else {
```

```
            printf("Input ERROR!\n");
            return 1;
      }
      return 0;
}

int GetStrLen(const char * str)
{
      int strLen = 0;
      while( (str[strLen] != '\0') && (str[strLen] != '\n') ) strLen++;
      return strLen;
}
```

Завдання 5. Напишіть на мові С функцію char * GetStrRevers(const char * src, char * dst) та програму, яка тестує її можливості. Функція здійснює реверс символів в рядку. Заборонено використовувати бібліотечні функції обробки рядків. Максимальна кількість символів у рядку не може перевищувати 512.

Приклад реалізації:

```
#include <stdio.h>
#define MAX_STR_LENGTH 512
char * GetStrRevers(const char * src, char * dst);
int main()
{
      char strInTest[MAX_STR_LENGTH] = "";
      char strOutTest[MAX_STR_LENGTH] = "";

      printf("Enter string:");
      if( fgets(strInTest, MAX_STR_LENGTH, stdin) != NULL )
      {
            int i = 0; while(strInTest[i] != '\n') i++;
            strInTest[i] = '\0';
            printf("Source String: %s\n", strInTest);
            printf("Revers String: %s\n", GetStrRevers(strInTest, strOutTest) );
      }
      else {
            printf("Input ERROR!\n");
            return 1;
      }

      return 0;
}

char * GetStrRevers(const char * src, char * dst)
{
      int strLen = 0;
      while( (src[strLen] != '\0') && (src[strLen] != '\n') ) strLen++;
      for(int i=0; i<strLen; i++) dst[i] = src[strLen-1-i];
      dst[strLen] = '\0';
      return dst;
}
```

Завдання 6. Написати на мові С програму, яка робить заміну у вхідному потоці всіх символів 'A' на символ 'a' та виводить результат на екран. Вхідний потік повинен бути пов'язаний з файлом на диску, який відкривається в режимі "тільки читання".

Приклад реалізації:

```
#include <stdio.h>
int main()
{
	FILE * fin = NULL;
	char inputFileName[512] = "";
	printf("Enter input file name:");
	scanf("%s", inputFileName);
	if( (fin=fopen(inputFileName, "r")) != NULL )
	{
		while(!feof(fin))
		{
			int ch = fgetc(fin);
			if(ch != EOF)
			{
				if(ch == 'A') fputc('a', stdout);
				else fputc(ch, stdout);
			}
		}
		printf("\n");
		fclose(fin);
	}
	return 0;
}
```

Завдання 7. Написати на мові С програму, яка робить заміну у вхідному потоці всіх спецсимволів '\t' на 8-м символів пробілу та виводить результат у потік виведення. Потоки повинні бути пов'язані з файлами на диску. Файл, що містить вхідні дані повинен відкриватися в режимі "тільки читання".

Приклад реалізації:

```
#include <stdio.h>
int main()
{
	FILE * fin = NULL;
	FILE * fout = NULL;
	char inputFileName[512] = "";
	char outputFileName[512] = "";
	printf("Enter input file name:"); scanf("%s", inputFileName);
	printf("Enter output file name:"); scanf("%s", outputFileName);
	if( (fin=fopen(inputFileName, "r")) != NULL )
	{
		if( (fout=fopen(outputFileName, "w")) != NULL )
		{
			while(!feof(fin))
			{
				int ch = fgetc(fin);
				if(ch != EOF)
				{
```

```
                        if(ch == '\t') for(int i=0;i<8;i++) fputc('\x20', fout);
                        else fputc(ch, fout);
                  }
            }
            fclose(fout);
      }
      fclose(fin);
   }
   return 0;
}
```

Завдання 8. Написати на мові С програму, яка підраховує кількість рядків в текстових файлах.

Приклад реалізації:

```
#include <stdio.h>
int main()
{
      FILE * fin = NULL;
      char inputFileName[512] = "";
      printf("Enter input file name:"); scanf("%s", inputFileName);
      if( (fin=fopen(inputFileName, "r")) != NULL )
      {
            int flagStartRow = 0;
            int numberLines = 0;
            int prevCh = 0;
            while(!feof(fin))
            {
                  int ch = fgetc(fin);
                  if(ch != EOF)
                  {
                        flagStartRow = 1;
                        if(ch == '\n') { numberLines++; flagStartRow = 0; }
                        prevCh = ch;
                  }
                  else { if(prevCh == '\n') numberLines++; } // New Empty Line!
            }
            numberLines = (flagStartRow)?numberLines+1:numberLines;
            fclose(fin);
            printf("Number Lines = %d\n", numberLines);
      }
      return 0;
}
```

Завдання 9. Напишіть на мові С програму, яка дозволяє виробляти ті ж дії, що і наступна команда в командному рядку Windows: copy con newfile.txt

Приклад реалізації:

```
#include <stdio.h>
#include <string.h>
void printHelp();
int main(int argc, char * argv[])
{
      FILE * fout = NULL;
```

```
	if(argc!=3)
	{
		printHelp();
		return 1;
	}

	if( strcmp("con", argv[1]) != 0 )
	{
		printHelp();
		return 2;
	}

	if( (fout=fopen(argv[2], "w")) != NULL )
	{
		while(!feof(stdin))
		{
			int ch = fgetc(stdin);
			if(ch != EOF) fputc(ch, fout);
		}
		fclose(fout);
	}
	return 0;
}

void printHelp()
{
	printf("parameter ERROR!\n");
	printf("Example: ex9 con newfile.txt\n");
}
```

Завдання 10. Написати функцію і програму на мові С. Написати програму, яка використовує функцію, що обчислює суму і різницю заданої кількості елементів масиву дійсних чисел. Функція визначена в головному файлі програми. Масив оголошений як локальний в функції main( ): float data[10];
Дані вводяться в консолі з потоку стандартного введення. Прототип функції:

int GetSumAndDiff(const float * pData, int count, int borderIndex,
float * pSum, float * pDiff);

Покажчик pData вказує на константні дані, які функція немає права змінювати; параметр count містить загальну кількість елементів масиву; параметр borderIndex вказує до якого елементу (включаючи і елемент з номером borderIndex) масиву повинна проводитися обробка; покажчики pSum та pDiff повертають в програму суму та різницю відповідно.

Функція повертає 1, якщо значення borderIndex знаходиться у певних межах та 0, якщо значення borderIndex помилкове.

Введення даних в масив здійснюється в основній програмі за допомогою функції scanf( ). Виведення результату здійснюється в основній програмі за допомогою функції printf( ).

Приклад реалізації:

```
#include <stdio.h>
#define MAX 10
int GetSumAndDiff(const float * pData, int count, int borderIndex,
            float * pSum, float * pDiff);
int main()
{
      float data[MAX] = {0};
      float sum = 0, diff = 0;
      for(int i=0; i<MAX; i++)
      {
            printf("Enter DATA[%d]:", i+1);
            scanf("%f", &data[i]);
      }

      if( GetSumAndDiff((const float *)data, MAX, 3, &sum, &diff) )
      {
            printf("Sum = %lf\nDiff = %lf\n", sum, diff);
      }
      else
      {
            printf("ERROR!\n");
      }
      return 0;
}

int GetSumAndDiff(const float * pData, int count, int borderIndex,
                  float * pSum, float * pDiff)
{
      if( (borderIndex>=0)&&(borderIndex<count) )
      {
            *pSum = 0;
            for(int i=0; i<=borderIndex; i++) *pSum += pData[i];
            *pDiff = pData[0];
            for(int i=1; i<=borderIndex; i++) *pDiff -= pData[i];
            return 1;
      }
      else return 0;
}
```

Завдання 11. Написати на мові С функцію float ** calculateMatrix(const float ** pSrcMatrix, int N, float ** pResultMatrix), яка в константній матриці pSrcMatrix розміром NxN ділить кожен елемент стовпця на значення, яке розташоване на головній діагоналі в тому ж стовпці, в якому йде перетворення і повертає матрицю pResultMatrix.

Приклад реалізації:

```
#include <stdio.h>
#include <malloc.h>
#define MATRIX_SIZE 4
float ** calculateMatrix(const float ** pSrcMatrix, int N,
                         float ** pResultMatrix);
```

```
int main()
{
	float ** matrix1 = (float**)malloc(sizeof(float*)*MATRIX_SIZE);
	for(int i=0; i<MATRIX_SIZE; i++)
		matrix1[i] = (float*)malloc(sizeof(float)*MATRIX_SIZE);

	float ** matrix2 = (float**)malloc(sizeof(float*)*MATRIX_SIZE);
	for(int i=0; i<MATRIX_SIZE; i++)
		matrix2[i] = (float*)malloc(sizeof(float)*MATRIX_SIZE);

	printf("SrcMatrix:\n");

	for(int i=0; i<MATRIX_SIZE; i++)
	{
		for(int j=0; j<MATRIX_SIZE; j++)
		{
			matrix1[i][j] = 1 + j + i; // Initialization of matrix
			printf("%.2f ", matrix1[i][j]);
		}
		printf("\n");
	}

	printf("==================\n");

	matrix2 = calculateMatrix((const float **)matrix1, MATRIX_SIZE, matrix2);

	printf("ResultMatrix:\n");

	for(int i=0; i<MATRIX_SIZE; i++)
	{
		for(int j=0; j<MATRIX_SIZE; j++)
		{
			printf("%.2f ", matrix2[i][j]);
		}
		printf("\n");
	}

	for(int i=0; i<MATRIX_SIZE; i++) free(matrix2[i]);
	free(matrix2);
	for(int i=0; i<MATRIX_SIZE; i++) free(matrix1[i]);
	free(matrix1);

	return 0;
}

float ** calculateMatrix(const float ** pSrcMatrix, int N,
                         float ** pResultMatrix)
{
	for(int i=0; i<N; i++) // rows
	  for(int j=0; j<N; j++) // cols
		{
			pResultMatrix[i][j] = pSrcMatrix[i][j] / pSrcMatrix[i][i];
		}
	return pResultMatrix;
}
```

Завдання 12. Написати на мові С функцію float ** calculateMatrix (const float ** pSrcMatrix, int N, float t, float k, float ** pResultMatrix), яка в матриці pSrcMatrix розміром NxN перемножує всі елементи, які розташовані на головній діагоналі на значення коефіцієнта t, а всі значення, які розташовані на побічній діагоналі, ділить на значення коефіцієнта k. Результат обчислення повертається в матриці pResultMatrix.

Приклад реалізації:

```
#include <stdio.h>
#include <malloc.h>
#define MATRIX_SIZE 4
float ** calculateMatrix(const float ** pSrcMatrix, int N,
                         float t, float k, float ** pResultMatrix);
int main()
{
      float ** matrix1 = (float**)malloc(sizeof(float*)*MATRIX_SIZE);
      for(int i=0; i<MATRIX_SIZE; i++)
            matrix1[i] = (float*)malloc(sizeof(float)*MATRIX_SIZE);

      float ** matrix2 = (float**)malloc(sizeof(float*)*MATRIX_SIZE);
      for(int i=0; i<MATRIX_SIZE; i++)
            matrix2[i] = (float*)malloc(sizeof(float)*MATRIX_SIZE);

      printf("SrcMatrix:\n");

      for(int i=0; i<MATRIX_SIZE; i++)
      {
            for(int j=0; j<MATRIX_SIZE; j++)
            {
                  matrix1[i][j] = 1 + j + i;
                  printf("%.2f ", matrix1[i][j]);
            }
            printf("\n");
      }

      printf("==================\n");

      matrix2 = calculateMatrix((const float **)matrix1, MATRIX_SIZE,
                              2., 3., matrix2);

      printf("ResultMatrix:\n");

      for(int i=0; i<MATRIX_SIZE; i++)
      {
            for(int j=0; j<MATRIX_SIZE; j++)
            {
                  printf("%.2f ", matrix2[i][j]);
            }
            printf("\n");
      }
```

```
	for(int i=0; i<MATRIX_SIZE; i++) free(matrix2[i]);
	free(matrix2);

	for(int i=0; i<MATRIX_SIZE; i++) free(matrix1[i]);
	free(matrix1);

	return 0;
}

float ** calculateMatrix (const float ** pSrcMatrix, int N,
                          float t, float k, float ** pResultMatrix)
{
	for(int i=0; i<N; i++) for(int j=0; j<N; j++)
		pResultMatrix[i][j] = pSrcMatrix[i][j];

	for(int i=0; i<N; i++) // rows
	  for(int j=0; j<N; j++) // cols
		{
			if(i == j) pResultMatrix[i][j] = pSrcMatrix[i][j] * t;
		}

	for(int i=0; i<N; i++) // rows
	  for(int j=N-1; j>=0; j--) // cols
		{
			if((N-1-i) == j) pResultMatrix[i][j] = pSrcMatrix[i][j] / k;
		}

	return pResultMatrix;
}
```

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Brian W. Kernighan, Dennis M. Ritchie. C Programming Language, 2nd Edition. – Pearson, 1988. – 272 p. ISBN-10: 0131103628, ISBN-13: 978-0131103627.

2. Clovis L. Tondo, Scott E. Gimpel. The C Answer Book: Solutions to the Exercises in 'The C Programming Language,' Second Edition 2nd Edition. – Pearson, 1988. – 208 p. ISBN-10: 0131096532, ISBN-13: 978-0131096530.

3. Алгоритмізація та програмування: Практикум [Електронний ресурс]: навч. посіб. для здобувачів ступеня бакалавра за спеціальністю 122 “Комп’ютерні на-уки” / Л.І. Кублій; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 28,15Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2019. – 209 с.

4. Вступ до програмування мовою С++. Організація обчислень: навч. посіб. / Ю. А. Бєлов, Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставровський. – К.: Видавничо-поліграфічний центр "Київський університет", 2012. – 175 с.

5. Грицюк Ю.І., Рак Т.Є. Програмування мовою С++: навчальний посібник. – Львів: Вид-во Львівського ДУ БЖД, 2011. – 292 с. ISBN 978-966-3466-85-9.

6. Васильєв О. Програмування на С++ в прикладах і задачах. – Ліра-К, 2017. – 382 с. ISBN 978-617-7507-41-2.

7. Шпак З.Я. Програмування мовою С. – Львів: Оріяна-Нова, 2006. – 432 с. ISBN 5-8326-0155-6.

8. Трофименко О.Г. С++. Алгоритмізація та програмування: підручник / О.Г. Трофименко, Ю.В. Прокоп, Н.І. Логінова, О.В. Задерейко. 2-ге вид. перероб. і доповн. – Одеса: Фенікс, 2019. – 477 с. ISBN 978-966-928-402-0.

9. Татарчук, Д.Д. Програмування мовами С та С++ [Електронний ресурс]: навчальний посібник / Д.Д. Татарчук, Ю.В. Діденко; НТУУ «КПІ». – Електронні текстові данні (1 файл: 949,75 Кбайт). – Київ : НТУУ «КПІ», 2012. – 112 с. – Назва з екрана.

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

Навчальне видання

Гаркуша Ігор Миколайович

Конспект лекцій
з дисципліни “Програмування”.
Для студентів галузі знань 12 “Інформаційні технології”
спеціальностей
123 “Комп’ютерна інженерія”,
126 “Інформаційні системи та технології”

Електронний ресурс

Видано
у Національному технічному університеті
«Дніпровська політехніка».
Свідоцтво про внесення до Державного реєстру ДК №1842 від 11.06.2004.
49005, м. Дніпро, просп. Дмитра Яворницького, 19.