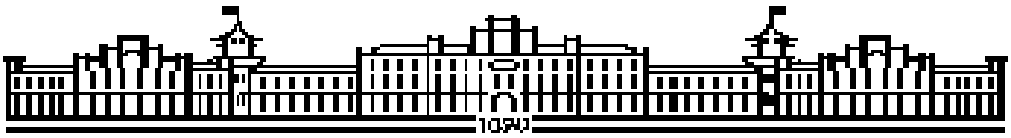


МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

**Конспект лекцій**
по курсу

# Алгоритми і структури даних

( назва дисципліни)

для напряму підготовки (спеціальностей)

123 Комп'ютерна інженерія .
(шифр та назва напряму, спеціальностей)

Уклали
Доцент каф. ОТ Анатолій Михайлович Сергієнко,
д.т.н., с.н.с.
доцент каф. СП і СКС Олександр Іванович Марченко,
к.т.н., доцент

***Рекомендовано***
***Вченою радою факультету***
***інформатики та обчислювальної***
***техніки НТУУ «КПІ»***
***Протокол № _ від __._____2017 р.***

Київ - 2017

# Лекція 1
# Вступ. Поняття алгоритму і структур даних

Щоб зрозуміти структуру існуючих комп'ютерів або успішно завершити розробку нових комп'ютерів, спілкуватися з експертами та зрозуміти відповідну технічну літературу, необхідно знати деякі визначення, аксіоми та принципи інформатики. Деякі з них наведено нижче.

**Стан комп'ютера** — це стан усіх його комірок пам'яті в дискретний момент часу його роботи. Наприклад, стан мікропроцесора визначається вмістом його, регістрів даних, команд, лічильником команд, триггерами прапорців, буферами інтерфейсів, а також усіма комірками оперативної пам'яті. Коли ми маємо справу з ручною моделлю процесора на аркуші паперу, то її порожні квадрати можуть служити комірками пам'яті, в яких запис виконується ручкою.

Різні **конструктивні об'єкти** можуть зберігатися в комірках пам'яті. Вони представляють окремі біти, символи, слова, рядки, числа, а також більш складні об'єкти та дані, тобто, **структури даних**, такі як записи, списки, файли, масиви тощо. Ці конструктивні об'єкти вказують стан комп'ютера відповідно до семантики обчислювального процесу, що виконується в ньому.

Конструктивні об’єкти мають задовольняти, принаймні, дві вимоги.

По-перше, кожен конструктивний об’єкт повинен надійно і однозначно відрізнятись від інших конструктивних об’єктів. Таким чином, конструктивні об’єкти правильно розпізнаються і обробляються у алгоритмах у будь-яких обставинах.

По-друге, мають бути чіткі правила побудови, конструювання, породження конструктивних об’єктів. Через це, власне, ці об’єкти і називаються конструктивними. Наприклад, якщо вже є такий об’єкт, як ціле число і, то за правилом породження, наступне число буде і+1.

**Обчислювальний процес** являє собою послідовність станів Sti комп'ютера, що починається з початкового стану St0 і закінчується результуючим, кінцевим станом StN. Крім того, результуючий стан може бути недосяжним, наприклад, у цифровому процесорі сигналу або в керуючому процесорі. У початковому стані певні комірки пам'яті зберігають початкові дані, а в кінцевому стані комірки зберігають результати розрахунків.

Під час обчислювального процесу конструктивні об'єкти зчитуються з пам'яті, перетворюються, розраховуються, пересилаються та зберігаються за допомогою певних операцій, інструкцій, що є специфічними для даного комп'ютера.

**Алгоритм** — це обчислювальний процес обчислення конкретної функції F в обчислювальній моделі, яка описується за допомогою строгих

математичних понять.

Це визначення було вперше сформульовано Е.Л. Постом і А.Т'юрінгом. У визначенні Т'юрінга взята машина Т'юрінга як модель і сам алгоритм. Ця машина побудована як нескінченна стрічка і блок керування. Головка читання може читати та записувати біти даних на стрічці. Блок керування контролює рух та роботу головки відповідно до алгоритму. Стан цієї моделі визначається станом блоку керування, положенням головки та станом стрічки і називається конфігурацією машини Тьюринга.

Поняття алгоритму існує як інтуїтивне поняття, а не як строго обмежене визначення. Це поняття передбачає, ***по-перше***, що повинен бути певний суб'єкт або процесор, який вміє читати, розпізнавати конструктивні об'єкти та правильно виконувати операції з ними відповідно до алгоритму.

***По-друге***, звичайно, алгоритм призначений для ефективного обчислення деякої корисної функції F для отримання правильного результату для певного виду вхідних даних. Крім того, досягнення такого результату має бути гарантованим за кінечну кількість кроків.

***По-третє***, для полегшення розуміння та реалізації алгоритму, незначні деталі обчислювального процесу (які зазвичай не відображені у обчислювальній моделі) не враховуються. На практиці, використовується набір різних обчислювальних моделей. Вони розрізняються як в області застосування, так і на рівні абстракції. В обох ситуаціях модель повинна бути доволі простою, щоб спрощувати організацію процесу обчислення.

Концепція конструктивних об'єктів також служить для того, щоб не розглядати незначні деталі алгоритму. Конструктивні об'єкти зазвичай утворюють набір абстрактних рівнів. Наприклад, якщо ми розглянемо обробку масиву, то ми можемо не розглядати окремі дані цих масивів, і, крім того, ми не маємо брати до уваги обробку окремих бітів цих даних.

Отже, концепція алгоритму нерозривно пов'язана з поняттям обчислювального процесу та обчислювальної моделі, які стосуються конструктивних об'єктів (див. Рис. 1.1). Коли ми говоримо про деякий обчислювальний процес, то ми вважаємо, що він відбувається в заданій обчислювальної моделі. Коли ми маємо належну обчислювальну модель, то ми можемо організувати у ній певний обчислювальний процес. І в обох цих ситуаціях ми маємо справу з алгоритмом, визначення якого залишається неточним.

Таким чином, алгоритм може існувати окремо від обчислювальної моделі — як моделі зі списком правил — або бути частиною моделі в якості пристрою керування в машині Тьюрінга або бути самою моделлю, наприклад, моделлю кінечного автомату або моделлю графу потоків даних. Наприклад, якщо виконавець алгоритму — це людина або програмований процесор, тоді алгоритм традиційно визначається як перелік інструкцій, які слід виконувати послідовно.

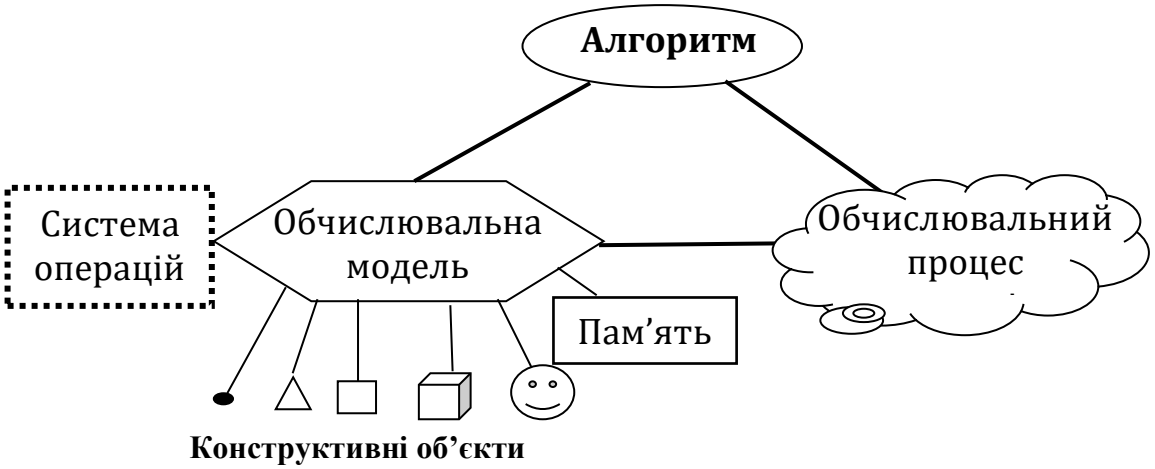



Рис.1.1 — Концепція алгоритму

Отже, ми можемо дати наступне визначення архітектури та інших визначень, пов'язаних з алгоритмами і структурами даних.

**Архітектура** — це модель реального комп'ютера з рівнем деталізації, яка є достатньою для його розробки або програмування, і забезпечує реалізацію відповідного набору алгоритмів.

**Програма** є алгоритмом, який розробляється для певної архітектури комп'ютера за допомогою алгоритмічної мови або машинних кодів в залежності від рівня деталізації архітектури або рівня абстракції.

Отже, програма, яка написана для певної архітектури, повинна бути обчислена для тих самих вхідних даних з однаковими проміжними та кінцевими результатами на різних комп'ютерах, що мають однакову архітектуру.

**Архітектура з точки зору розробника** — це комп'ютерна модель з рівнем деталізації, яка є достатньою для його розробки та виробництва. Архітектура, як правило, включає в себе інформацію про набір інструкцій, адресний простір, управління адресами, системи переривання, захист пам'яті, інтерфейси, периферійні пристрої, тобто всю інформацію про детальне технічне завдання для розробки комп'ютера. Опис цієї архітектури не включає інформацію про елементну базу, її швидкодію, розміри комп'ютера, споживання електроенергії, параметри безпеки тощо, оскільки вони безпосередньо не впливають на обчислювальний процес.

**Архітектура з точки зору програміста** — це комп'ютерна модель з рівнем деталізації, який є достатнім для успішного програмування певних обчислювальних задач на певній алгоритмічній мові.

Наприклад, програміст на мові Паскаль враховує модель пам'яті комп'ютера, до якої звертаються по 32-розрядній шині, з ALU, який обробляє цілі числа чи числа з плаваючою точкою, з пам'яттю довільною ємністю на жорсткому магнітному диску, клавіатуру та дисплей, доступ до яких забезпечується процедурами, які програміст може знайти в бібліотеці процедур Паскаля.

Кожний рядок алгоритму, який описаний мовою асемблера, означає конкретну машинну інструкцію, яка виконує елементарну операцію з даними або керує вибором наступної інструкції. Тому програмісту на мові асемблера потрібно добре знати архітектуру комп'ютера в деталях до окремого регістра та його біту, коду переривання, адреси периферійного пристрою тощо.

Деякі архітектури для мови високого рівня визначаються як інтерфейс між мовою та системним програмним забезпеченням, яке безпосередньо реалізує скомпільовану програму. Як такий інтерфейс, як правило, виступає операційна система або, так звана, віртуальна машина.

**Архітектурна платформа** — це загальна комп'ютерна архітектура, яка є гарантовано незмінною протягом наступних кількох років або навіть десятиріч. Архітектурна платформа може використовуватися в нових обчислювальних інструментах та на комп'ютерах, забезпечуючи сумісність програмного забезпечення, використання різних компонентів в комп'ютерах, підключення існуючих периферійних пристроїв та пристроїв, вироблених іншими компаніями. Найвідоміша архітектурна платформа — це i80x86. Її синонім — 32-розрядна архітектура Intel (IA-32).

**Архітектурна парадигма** являє собою набір загальних принципів і підходів для проектування комп'ютерних архітектур. Наприклад, комп'ютер ENIAC був зроблений за допомогою спеціальної парадигми процесора, яка використовує відображення алгоритму графа в структуру процесора з налаштовуваними з'єднаннями. Тобто, вершина графу асоціюється з операційним блоком, тобто суматором або блоком множення, а її ребро — вказує зв'язок між операційними блоками. Більшість комп'ютерів, якими користуються люди, побудовані за парадигмою фон Неймана, що буде розглянута нижче.

# Лекція 2.
# Архітектура комп’ютера і мови програмування

## Архітектурна ієрархія

Через принцип ієрархії у сучасних комп'ютерах розглядаються два або більше рівнів архітектури. У комп'ютерах існують принаймні п'ять рівнів ієрархії архітектур, як показано на рис. 1.2.

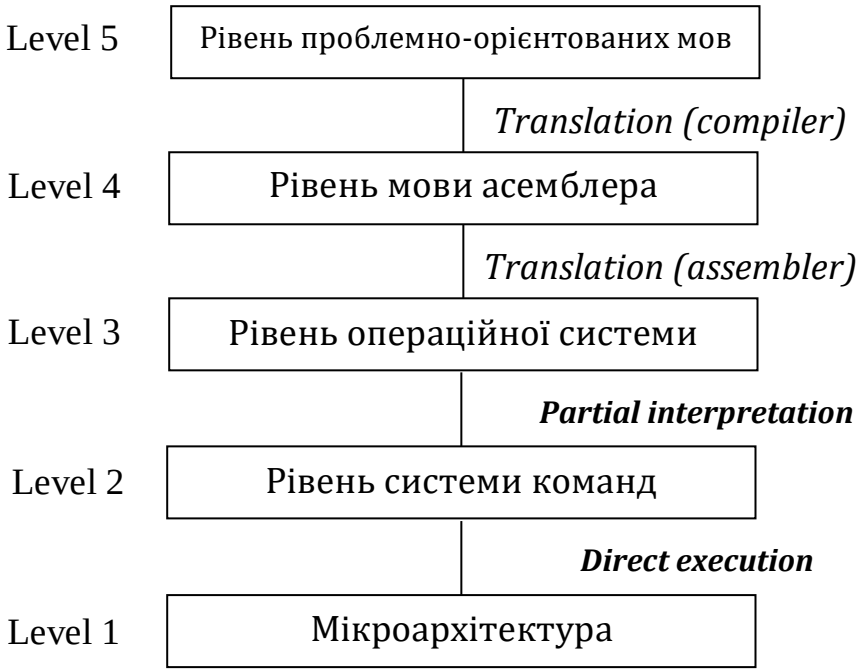



Рис.1.2. Ієрархічні рівні архітектури

**Рівень мікроархітектури** розглядається як нижній рівень архітектури 1. Він описує деталі реалізації архітектури апаратури. Вони полягають у тому, як арифметичні та логічні інструкції виконуються в ALU, внутрішня структура регістрового ОЗП, як шини з’єднують блоки процесора, і які сигнали управління керують записом даних у регістри і так далі.

Кожен процесор має свою мікроархітектуру. Отже, мікроархітектура може оцінюватися як своєрідна архітектура досить відносно, оскільки вона залежить від елементного базису.

Архітектура рівня 2 називається **архітектурою на рівні системи команд** (рівень ISA). Кожен виробник процесорів публікує посібник для кожного процесора під назвою «Довідник з машинної мови», або щось подібне. Ці посібники дійсно стосуються рівня ISA, а не мікроархітектури. Крім того, рівень 1, як правило, є секретною інформацією. Коли такий посібник описує набір інструкцій процесора, він насправді описує інструкції, що виконуються інтерпретуючими внутрішніми мікропрограмними або апаратними схемами.

Наступний рівень – **рівень операційної системи** – як правило, є гібридним рівнем. Більшість інструкцій у цьому рівні належать також до рівня ISA. Крім того, існує множина інструкцій, які стосуються лише операційної

системи. У рівня інша організація пам'яті, включаючи захист пам'яті, можливість одночасного запуску двох або більше програм та різних інших функцій.

**Операційна система** (ОС) - це набір програм, що дозволяє користувачеві ефективно організувати виконання різних завдань обчислень на комп'ютері та тримати комп'ютерну систему у робочому стані. Основними завданнями ОС, що працюють на рівні 3, є завантаження користувацької програми та даних у пам'ять, початок роботи, забезпечення паралельної реалізації декількох програм у режимі розподілу часу, запуск та відпрацювання винятків. Спеціальні команди та апаратні засоби, які недоступні безпосередньо з програм користувача, підтримують ці завдання.

Рівень 4, **рівень мови асемблера**. Цей рівень дає програмістам спосіб писати програми для рівнів 2 і 3 у формі, яка не настільки неприємна, як сама машинна мова. Програми на ассемблері спочатку переводяться на рівень 2 або 3 мови, а потім інтерпретуються відповідною віртуальною чи фактичною машиною.

**Рівень мови користувача**, як правило, складається з мов, призначених для використання програмами-застосунками. Такі мови часто називаються мовами високого рівня. Деякі з найбільш відомих є C, C ++, C #, Java, Pascal, Python і PHP. Програми, написані цими мовами, зазвичай транслюються на 3-й або 4-й рівень компіляторами.

Таким чином, головне, що слід пам'ятати, це те, що комп'ютери розроблені як набір архітектурних рівнів, кожний з яких побудований на попередніх рівнях. Кожен рівень представляє собою чітко виражену абстракцію, в якій присутні різні об'єкти та операції. Розробляючи та аналізуючи комп'ютери таким чином, ми можемо не приймати до уваги архітектурні деталі і, таким чином, зменшити складність предмету.

## Модель фон Неймана

Типове програмно кероване ядро комп'ютера, який функціонує за принципом фон-Неймана, складається з блока керування (БК) – керуючого автомата і блока обробки даних (БОД) – операційного автомата. БК забезпечує вибірку й декодування команд і видачу відповідних керуючих сигналів. В свою чергу, БОД складається з трьох груп схем: блока регістрів даних, арифметико-логічного пристрою (АЛП) і блока локальної пам'яті (див. рис. 1.6). Часто ядро комп'ютера називають центральним процесорним елементом (ЦПЕ).

Блок обробки даних забезпечує виконання кожної команди в ядрі процесора. Блок регістрів даних – це група регістрів загального призначення, які зберігають дані для обчислень при виконанні команд. АЛП виконує всі арифметичні і логічні операції, а також операції зсуву. Блок локальної пам'яті зберігає дані й програми та забезпечує їх швидку вибірку. В сучасних процесорах – це асоціативний оперативний запамятовуючий пристрій (ОЗП), тобто, кеш-ОЗП.

Функціонування моделі комп'ютера фон-Неймана ґрунтується на повторенні чотириетапної процедури виконання команди.

На **першому етапі**, який називається вибірка команди, ядро процесора посилає до зовнішнього ОЗП адресу команди з лічильника команд (ЛК) та сигнал, що потрібна наступна команда. ОЗП відповідає тим, що висилає цю команду, яка попадає в блок керування, а саме – в регістр команди (РК).

На **другому етапі** – етапі декодування – блок керування вирішує, яка це команда та які дії треба зробити для її виконання. Одержана інформація пересилається в блок обробки даних.

**Третій етап** – це етап виконання команди. Блок обробки даних одержує необхідні дані з блоку регістрів даних, або з локальної пам'яті, або з зовнішнього ОЗП, обробляє їх і видає результати.

**Четвертий етап** – етап запам'ятовування результатів в локальній пам'яті або зовнішньому ОЗП. При цьому також визначається адреса наступної команди, як правило, в регістрі ЛК.

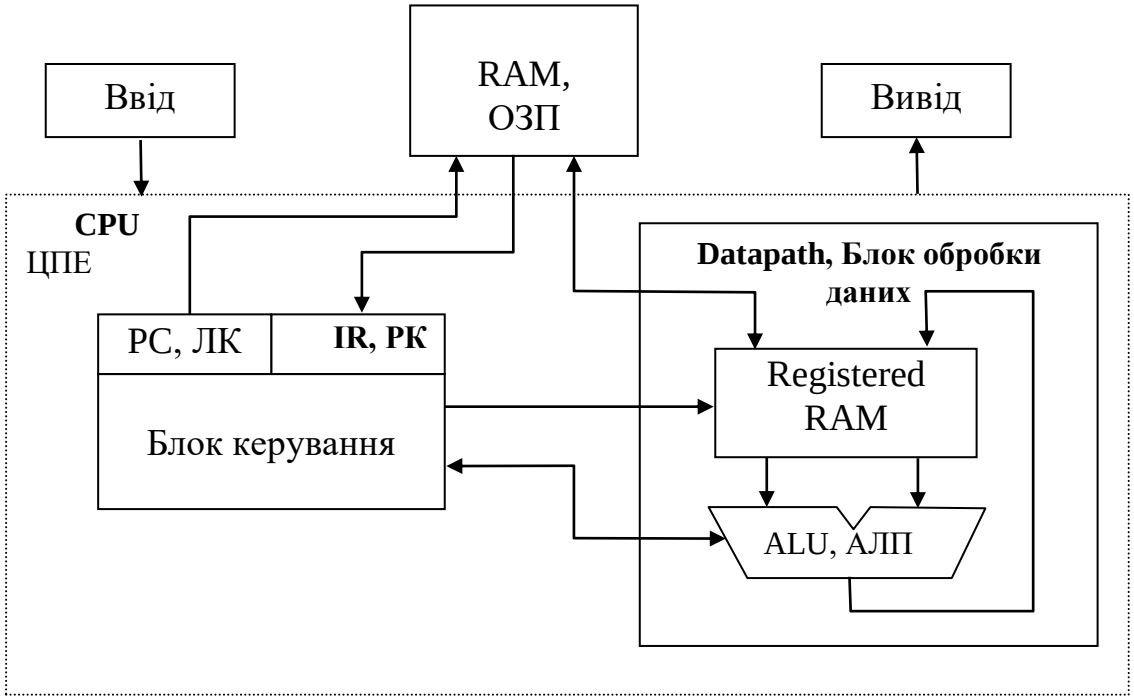



Рис.2.1. Архітектура фон Неймана

Кожна команда виконується за такою самою послідовністю етапів. Ядро процесора повторює ці чотири етапи під час виконання всієї програми. Відмінність лише в тому, що різні команди неоднаково керують блоком обробки даних та вибіркою даних і наступної команди.

# Лекція 3
# Основи мови Сі

## Конструктивні об’єкти мови СІ

1) Коментарі. Вони підвищують наочність і зручність читання програм. Коментарі обрамляються символами / * і * /. У мові Сі ++ введена ще одна форма запису коментарів. Все, що знаходиться після знаків // до кінця поточного рядка, буде також розглядатися як коментар.

Коментарі відаляються компілятором. Також видаляються або ігноруються пробіли, символи табуляції і переходу на новий рядок.

**2) Символи**

— рядкові та прописні латинскі букви: a ... z; A ... Z;
— цифри: 0 1 2 3 4 5 6 7 8 9;
— спеціальні символи: . , : ; ' " # { } [ ] ( ) < > & | ^ ! _ + - * / \ % = ? ~ ;
— символ "пробіл";
— нулевий символ чи "пусто" (NULL).

Деякі з символів можуть позначатися спеціальним чином:

— \? — знак питання;
— \' — апостроф;
— \" — лапки;
— \\ — зворотна похила риска;
— \0 — пусто.

Крім того, кожен символ може бути позначений власним кодом в шістнадцятковій системі числення:

— \xdd — де букви dd позначають код символа.

Додатково використовуються керуючі символи. Це такі символи, при вставці яких в текст виконується певна дія:

— \b — повернення на крок;
— \f — перехід на наступну сторінку;
— \n — перехід на наступний рядок;
— \r — перехід на першу позицію наявного рядка;
— \t — горизонтальна табуляція;

Для представлення кожного символа в компьютері використовується **один байт**, тому загальне число символів дорівнює $2^8$= 256.

**3) Ключові слова**

Ключові слова — це такі слова, які мають визначене призначення в цій мові і не можуть використовуватися для інших цілей. Нижче перераховані всі ключові слова мови Сі:

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

Компілятор замінює ключовими словами внутрішнім кодом — цілим числом.

**4) ідентифікатор**, який використовується в якості імені об'єкта (функції, змінної, константи і ін.). Ідентифікатори повинні вибиратися з урахуванням наступних правил:

1. Вони повинні починатися з літери латинського алфавіту (a, ..., z, A, ..., Z) або з символу підкреслення (_).
2. У них можуть використовуватися літери латинського алфавіту, символ підкреслення і цифри (0, ..., 9). Використання інших символів в ідентифікаторах заборонено.
3. У мові Сі букви нижнього регістра (a, ..., z), що застосовуються в ідентифікаторах, відрізняються від букв верхнього регістру (A, ..., Z). Це означає, що імена з одним прочитанням вважаються різними: name, Name, NAME і т.д.
4. Ідентифікатори можуть мати будь-яку довжину, але сприймається і використовується для розрізнення об'єктів (функцій, змінних, констант і т.д.) тільки частина символів. Їх число змінюється для різних систем програмування, але відповідно до стандарту ANSI C воно не перевищує 32.
5. Ідентифікатори не повинні збігатися з ключовими словами мови і іменами стандартних функцій з бібліотек.

Компілятор збирає ідентіфікатори у таблиці ідентіфікаторів, де кожному з них, в решті решт, призначає адресу в ОЗП. Також ця таблиця вміщує додаткову інформацію — тип об'єкта ...

Отже, кожному з ідентіфікаторів відповідає адреса комірки пам'яті, де зберігається відповідний об'єкт або місце початку цього об'єкту (початок масиву, рядка, підпрограми тощо). Тому Ідентифікатор називають також **символьною адресою**.

5) **Літерали**

Константами або літералами називаються деякі фіксовані значення даних, тобто, такі значення, які не можуть змінюватися.

Розрізняють чотири типи літералів:

— цілі літерали;

— плаваючі літерали;

— символьні літерали;

— рядкові літерали.

**Цілий літерал** може бути записаний в десятковій, вісімковій або шістнадцятковій системі числення. У десятковій системі цілий літерал записується як звичайне десяткове число, за умови, що перша цифра не є нулем.

У шістнадцятковій системі числення цілий літерал записується шістнадцатковими цифрами і повинен починатися з символів 0x або 0X. При цьому для позначення шістнадцяткових цифр від 10 до 15 можуть використовуватися як малі літери a, b, c, d, f, так і великі літери A, B, C, D, F. Наприклад, такі цілі числа є цілими шістнадцятковими літералами:

0x12, 0X120xABC, 0Xавс

Крім того, в мові програмування Сі дозволяється оголошення довгих цілих літералів. Для цієї мети в кінці літерала ставиться буква l або L.

**Літерал з плаваючою точкою** представляє деяке дійсне число і має такий вигляд:

[ціла частина].[дробна частина][E|e[+|-]експонента]

У визначенні літерала з плаваючою точкою має бути присутня, принаймні, одна з частин, укладених в квадратні дужки. Нижче наведені приклади літералів з плаваючою точкою:

3., .14, 3.14, 0.314e1, 314e-2.

**Символьний літерал** складається з одного символу, який полягає в апострофи. Нижче наведені приклади символьних літералів:

'C', 'y', '5', '\101'

Сам символ апостроф, який використовується в якості символьної константи, потрібно позначати як \'.

**Рядковий літерал** являє собою послідовність символів, укладену в лапки. За стандартом, довжина рядкового літерала не може перевищувати 509 символів. Нижче наведені приклади рядкових літералів.

"This is a string.", "Це рядок." , "A", "1"

Сам символ лапки, використовуваний в рядковій константі, потрібно позначати як \". Відзначимо, що в кінці кожного рядкового літерала компілятор поміщає нульовий символ \0, який відзначає кінець рядка.

Компілятор заміняє літерали відповідним машинним поданням, наприклад, символьний літерал — байтом, цілий літерал — машинним словом.

# Лекція 4.
# Дані та їхні типи.
# Оерації та оператори.

## Дані та їх Типи

Програми оперують з різними даними, які можуть бути простими і структурованими.

**Прості дані** — це цілі числа і числа з плавючою точкою, символи та покажчики (адреси об'єктів в пам'яті).

**Структуровані дані** — це масиви і структури; вони будуть розглянуті нижче.

У мові розрізняють поняття "тип даних" і "модифікатор типу".

Тип даних — це, наприклад, цілий, а модифікатор типу позначає — зі знаком або без знаку даний операнд. Ціле зі знаком матиме як позитивні, так і негативні значення, а ціле без знака — тільки позитивні значення.

У мові Сі можна виділити п'ять базових типів, які задаються наступними ключовими словами:

· char — символьний;
· int — цілий;
· float — реальний;
· double — реальний подвійної точності;
· void — без значення.

Дамо їм коротку характеристику:

1. Змінна типу char має розмір 1 байт, її значеннями є різні символи з кодовою таблиці ASCII, наприклад: 'ф', ':', 'j' (під час запису в програмі вони вміщуються в одинарні лапки).

2. Розмір змінної типу int в стандарті мови Сі не визначений. У більшості систем програмування розмір змінної типу int відповідає розміру цілого машинного слова.

3. Змінна типу float займає в пам'яті 32 біта. Вона може приймати значення в діапазоні від 3.4e-38 до 3.4e + 38.

4. Ключове слово double дозволяє визначити речову змінну подвійної точності. Вона займає в пам'яті 8 байтів.

5. Ключове слово void (який не має значення) використовується для видалення значення об'єкта, наприклад, для оголошення функції, яка не повертає ніяких значень.

У стандарті ANSI мови Сі є наступні **модифікатори типу**:

· unsigned
· signed
· short
· long

Модифікатори записуються перед специфікаторами типу, наприклад: unsigned char. Якщо після модифікатора опущений специфікатор, то компілятор припускає, що цим специфікатором є int. Таким чином, наступні рядки:

```
long a;
long int a;
```

є ідентичними і визначають об'єкт a як довгий цілий. Табл. 1 ілюструє можливі поєднання модифікаторів (unsigned, signed, short, long) зі специфікаторами (char, int, float і double), а також показує розмір і діапазон значень об'єкта (для 16-розрядних компіляторів).

Таблица 1

| Тип | Розмір в байтах (бітах) | Інтервал представлення |
|---|---|---|
| char | 1 (8) | от -128 до 127 |
| unsigned char | 1 (8) | від 0 до 255 |
| signed char | 1 (8) | від -128 до 127 |
| int | 2 (16) | від -32768 до 32767 |
| unsigned int | 2 (16) | від 0 до 65535 |
| signed int | 2 (16) | від -32768 до 32767 |
| short int | 2 (16) | від -32768 до 32767 |
| unsigned short int | 2 (16) | від 0 до 65535 |
| signed short int | 2 (16) | від -32768 до 32767 |
| long int | 4 (32) | від -2147483648 до 2147483647 |
| unsigned long int | 4 (32) | від 0 до 4294967295 |
| signed long int | 4 (32) | від -2147483648 до 2147483647 |
| float | 4 (32) | від 3.4E-38 до 3.4E+38 |
| double | 8 (64) | від 1.7E-308 до 1.7E+308 |
| long double | 10 (80) | від 3.4E-4932 до 3.4E+4932 |

**Змінні**

Всі змінні до їх використання повинні бути визначені (оголошені). При цьому задається тип, а потім йде список з однієї або більше змінних цього типу, розділених комами. Наприклад:

```
int a, b, c;
char x, y;
```

У мові розрізняють поняття оголошення змінної і її визначення.

Оголошення встановлює властивості об'єкта: його тип (наприклад, цілий), розмір (наприклад, 4 байта) і т.д.

Визначення поряд з цим викликає виділення пам'яті (в наведеному прикладі дано визначення змінних).

Змінні в мові Сі можуть бути ініційовані при їх визначенні:

```
int a = 25, h = 6;
char g = 'Q', k = 'm';
float r = 1.89;
long double n = r * 123;
```

**Локальні та глобальні об'єкти**

**Глобальні об'єкти** доступні для будь-якого оператора і функції.

**Локальні об'єкти** по відношенню до функцій є внутрішніми. Вони починають існувати, при вході в функцію і знищуються після виходу з неї.

Відзначимо, що виконання програми завжди починається з виклику функції main (), яка містить тіло програми. Тіло програми, як і тіло будь-якої іншої функції, поміщається між фігурними дужками {}, які відкриваються і закриваються.

У мові Сі усы визначення повинні слідувати перед операторами, які складають тіло функції.

**Операції мови Сі**

Будь-які **вирази** мови складаються з операндів (змінних, констант та ін.), які з'єднані знаками операцій.

**Знак операції** — це символ або група символів, які повідомляють компілятору про необхідність виконання певних арифметичних, логічних або інших дій.

Операції виконуються в строгій послідовності. Величина, що визначає переважне право на виконання тієї чи іншої операції, називається **пріоритетом**. У табл. 2 перелічені різні операції мови Сі. Їх пріоритети для кожної групи однакові (групи виділені). Чим більшою перевагою користується відповідна група операцій, тим вище вона розташована в таблиці. Порядок виконання операцій може регулюватися за допомогою круглих дужок.

Для виключення плутанини в поняттях "операція" та "оператор", відзначимо, що **оператор** — це найменша виконувана одиниця програми. Розрізняють оператори виразу, дія яких полягає в обчисленні заданих виразів (наприклад: a = sin (b) + c; j ++;), оператори оголошення, складені оператори, порожні оператори, оператори мітки, циклу і т.д. Для позначення кінця оператора в мові Сі слід ставити крапку з комою.

Що стосується **складеного оператора** (або блоку), що представляє собою набір логічно пов'язаних операторів, поміщених між відкриваючою ({) і закриваючою (}) фігурними дужками ( "операторними дужками"), то за ним крапка з комою не ставиться. Відзначимо, що **блок** відрізняється від складеного оператора наявністю визначень в тілі блоку.

Охарактеризуємо основні операції мови Сі. Спочатку розглянемо одну з них - операцію присвоювання (=). вираз виду

```
x = y;
```

присоює змінній x значення змінної y. Операцію "=" дозволяється використовувати багаторазово в одному виразі, наприклад:

```
x = y = z = 100;
```

Розрізняють **унарні і бінарні операції**. У перших з них один операнд, а у других — два. Почнемо їх розгляд з операцій, віднесених до першої з наступних традиційних груп:

1. Арифметичні операції.
2. Логічні операції і операції відношення.

3. Операції з бітами.

Таблиця 2 Операції мови Сі та їх пріоритети

| Пріоритет | Операція | Опис | Асоціативність |
|---|---|---|---|
| **1** (найвищий) | `++ --` | Суфіксні/постфіксні інкремент та | Зліва направо |
| | `()` | Виклик функції, дужки | |
| | `[]` | Звертання до елементу масиву | |
| | `.` (крапка) | Звертання до елементу структури чи | |
| | `–>` | Звертання до елементу структури через | |
| | (*type*){*list* | Складений літерал | |
| **2** | `++ --` | Префіксні інкремент та декремент | Справа наліво |
| | `+ –` | Унарні плюс та мінус | |
| | `! ~` | Логічне НІ та побітове НІ | |
| | (*type*) | Перетворення типу | |
| | * | Розіменування (*y – дане за адресою y) | |
| | & | Взяття адреси (&x – адреса даного x) | |
| | `sizeof` | Розмір об’єкту | |
| **3** | `* / %` | Множення, ділення і остача | Зліва направо |
| **4** | `+ –` | Додавання і віднімання | |
| **5** | `<< >>` | Побітові лівий та правий зсув | |
| **6** | `< <= > >=` | Оператори порівняння типу більше — | |
| **7** | `== !=` | Оператори порівняння = і ≠ | |
| **8** | & | Побітове І | |
| **9** | ^ | Побітове XOR | |
| **10** | \| | Побітове АБО | |
| **11** | && | Логічне І | |
| **12** | \|\| | Логічне АБО | |
| **13** | `?:` | Тернарна умова | справа наліво |
| **14** | `=` | Просте присвоювання | |
| | `+= –=` | Присвоювання через суму чи різницю | |
| | `*= /= %=` | Присвоювання через добуток, ділення чи | |
| | `<<= >>=` | Присвоювання через лівий чи правий | |
| | `&= ^= \|=` | Присвоюання через побітові І, виключне | |
| **15** | , | Кома | Зліва направо |

Приклад асоціативності зліва направо:
вираз `a*b/c` розбирається як (`a*b)/c`, а не як `a*(b/c)` .
Приклад асоціативності справа наліво:
вираз `a=b=c` розбирається як `a=(b=c)`, а не як `(a=b)=c` .

Арифметичні операції задаються символами (табл. 2): +, -, *, /, %. Останню з них можна використовувати лише до змінних цілого типу. Наприклад:

```
a = b + c;
x = y - z;
r = t * v;
s = k / l;
p = q % w;
```

Логічні операції відношення задаються наступними символами (див. Табл. 2): && ("І"), || ("АБО"), ! ("НІ"), > , > =, <, <=, = = (дорівнює), ! = (Не дорівнює). Традиційно ці операції повинні давати одне з двох значень: істину або неправду. У мові Сі прийнято наступне правило: істина — це будь-яке ненульове значення; неправда — це нульове значення. Вирази, що використовують логічні операції і операції відношення, повертають 0 для помилкового значення і 1 для істинного. Нижче наводиться таблиця істинності для логічних операцій.

Таблиця 3

| x | y | x&&y | x\|\|y | !x |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

Бітові операції можна застосовувати до змінних, які мають типи int, char, а також їх варіантів (наприклад, long int). Їх не можна застосовувати до змінних типів float, double, void (або більш складних типів). Ці операції задаються наступними символами: ~ (поразрядное заперечення), << (зсув вліво), >> (зсув вправо), & (порозрядне "І"), ^ (порозрядне виключне "АБО"), | (Порозрядне "АБО").

Приклади: якщо a = 0000 1111 и b = 1000 1000, то

```
    ~a = 1111 0000,
a << 1 = 0001 1110,
a >> 1 = 0000 0111,
a & b  = 0000 1000,
a ^ b  = 1000 0111,
a | b  = 1000 1111.
```

У мові передбачені дві нетрадиційні операції інкремента (++) і декремента (--). Вони призначені для збільшення і зменшення на одиницю значення операнда. Операції “++” і “—“ можна записувати як перед операндом, так і після нього. У першому випадку (++ n або --n) значення

операнда (n) змінюється перед його використанням у відповідному виразі, а в другому (n ++ або n--) — після його використання. Розглянемо два наступні рядки програми:

```
a = b + c ++;
a1 = b1 + ++ c1;
```

Припустимо, що b = b1 = 2, c = c1 = 4. Тоді після виконання операцій: a = 6, b = 2, c = 5, a1 = 7, b1 = 2, c1 = 5.

Широке поширення знаходять також вирази з ще однією нетрадиційною **тернарною** або умовною операцією?:. У операторі

```
y = x? a: b; // a при x = 1, b при x = 0
```

y = a, якщо x не дорівнює нулю (тобто істинно), і y = b, якщо x дорівнює нулю (неправда). Наступне вираз

```
y = (a> b)? a: b;
```

дозволяє привласнити змінній y значення більшої змінної (a чи b), тобто y = max (a, b).

Ще однією відмінністю мови є те, що вираз виду a = a + 5; можна записати в іншій формі: a + = 5 ;. Замість знака + можна використовувати і символи інших бінарних операцій (див. Табл. 2).

**Перетворення типів**

Якщо у виразі з'являються операнди різних типів, то вони перетворюються до деякого загального типу, при цьому до кожного арифметичному операнду застосовується така послідовність правил:

1. Якщо один з операндів у виразі має тип long double, то інші теж перетворюються до типу long double.
2. В іншому випадку, якщо один з операндів у виразі має тип double, то інші теж перетворюються до типу double.
3. В іншому випадку, якщо один з операндів у виразі має тип float, то інші теж перетворюються до типу float.
4. В іншому випадку, якщо один з операндів у виразі має тип unsigned long, то інші теж перетворюються до типу unsigned long.
5. В іншому випадку, якщо один з операндів у виразі має тип long, то інші теж перетворюються до типу long.
6. В іншому випадку, якщо один з операндів у виразі має тип unsigned, то інші теж перетворюються. до типу unsigned.
7. В іншому випадку всі операнди перетворюються до типу int. При цьому тип char перетворюється в int зі знаком; тип unsigned char в int, у якого старший байт завжди нульовий; тип signed char в int, у якого в знаковий розряд передається знак з char; тип short в int (знаковий або беззнаковий).

Припустимо, що обчислено значення деякого виразу в правій частині оператора присвоювання. У лівій частині оператора присвоювання записана деяка змінна, причому її тип відрізняється від типу результату в правій частині. Тут правила перетворення дуже прості: значення праворуч від оператора присвоювання перетвориться до типу змінної зліва від оператора

присвоювання. Якщо розмір результату в правій частині більше розміру операнда в лівій частині, то старша частина цього результату буде втрачена.

У мові Сі можна явно вказати тип будь-якого виразу. Для цього використовується операція перетворення ( "приведення") типу. Вона застосовується в такий спосіб:

```
(Тип) вираз
```

(Тут можна вказати будь-який тип, який допустимий в мові Сі).

Розглянемо приклад для машини з 16-розрядними цілими числами:

```
int a = 30000;
float b;
.......
b = (float) a * 12;
```

(Змінна a цілого типу явно перетворена до типу float; якщо цього не зробити, то результат буде втрачено, тому що a * 12 > 32767).

Перетворення типу також може використовуватися для перетворення типів аргументів при виклику функцій.

# Лекція 5
# Конструкції для створення алгоритмів

| Конструкції для створення алгоритмів | | |
|---|---|---|
| Конструкції блочності | Конструкції опису даних | Керуючі конструкції |

**Конструкції блочності**:

1. Програма
2. Процедура
3. Функція
4. Блок (програмний)
5. Модуль

**Блок** — виділена частина програми, змінні якої мають локальну дію, змінні, об'явлені у блоці є невидимимі ззовні блоку.

**Модуль** – виділена частина програми, яка може бути замінена на еквівалентну частину без зміни решти програми, часто – бібліотечні функції, сервіси, класи.

Вимоги до модуля:

— простота (Keep It Simple, Stupid, реалізує 1 функцію задачі),
— замкненість (незалежність, 1 вхід, 1 вихід) ,
— осяжність ( 50 — 100 рядків тексту),
— наявність інтерфейсу (щоб будь-хто міг користуватись модулем),
— приховання деталей реалізації.

**Керуючі конструкції**

I конструкції розгалуження

1) умовна конструкція (або одне, або інше)
   а) неповна умовна конструкція
   б) повна умовна конструкція
2) конструкція вибору варіанту (альтернативи)
   а) неповна конструкція
   б) повна конструкція

II конструкції повторення

1) цикл з передумовою
2) цикл з післяумовою
3) цикл з лічильником (з параметром)
4) цикл безумовний (нескінченний)

III Конструкції переходу

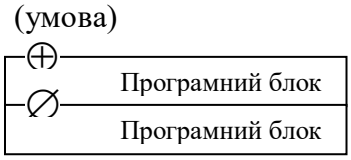
1) безумовний перехід (операція goto)
2) конструкція виходу з циклу
   а) однорівнева
   б) багаторівнева
3) конструкція продовження циклу
   а) однорівнева
   б) багаторівнева
4) конструкція виходу з блоку

## I конструкції розгалуження

**умовна конструкція** (або одне, або інше)

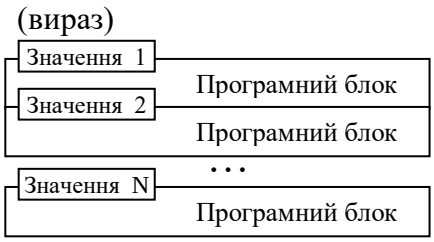
а) неповна умовна конструкція


повна умовна конструкція




**конструкція вибору варіанту**

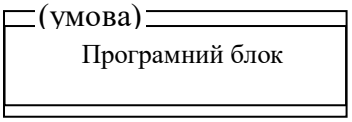
неповна конструкція



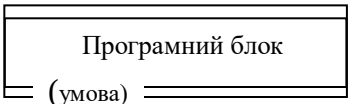

повна конструкція


## II конструкції повторення

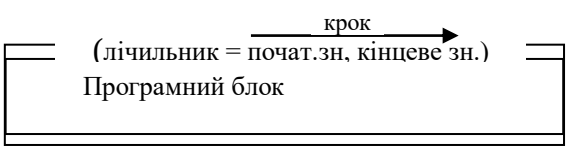
цикл з передумовою

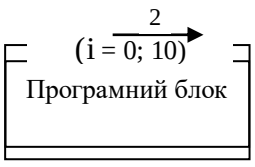


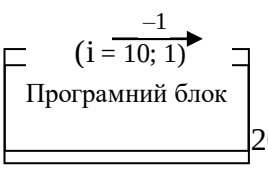

цикл з післяумовою




цикл з лічильником (з параметром)



цикл безумовний (нескінченний)

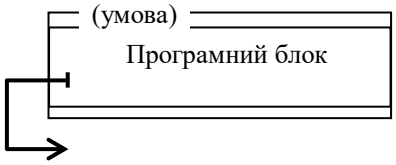

**Конструкції переходу**

1) Безумовний перехід (операція goto)

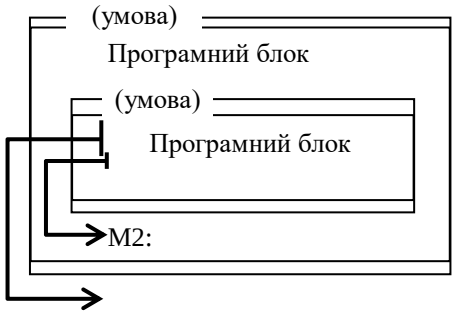
У діаграмах дій – не використовується.

2) конструкція виходу з циклу

а) одноріvнева



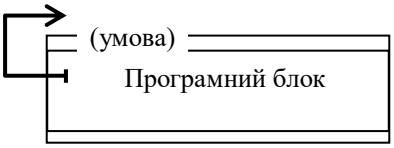

б) багаторівнева



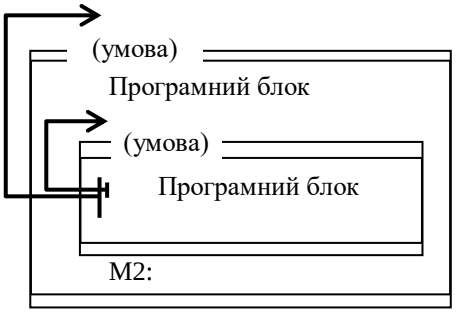

M1:

3) конструкція продовження циклу

а) одноріvнева




б) багаторівнева




4) конструкція виходу з блоку

# Лекція 6
# UML I Діаграма діяльності
# Типи алгоритмів за структурою

На діаграмі діяльності буде показано послідовність актів дій системи на основі Діяльностей. Діаграми діяльності є особливою формою діаграм стану, на яких містяться лише (або головним чином) діяльності.

Діаграми діяльності завжди пов'язано з *класом*, *операцією* або *випадком використання*.

На діаграмах діяльності може бути показано як послідовні, так і паралельні діяльності. Паралельне виконання буде показано за допомогою піктограм Розділити/Чекати, для діяльностей, які виконуються паралельно, неважливим є порядок їх обробки (їх може бути виконано одночасно або одну за одною).

**Діяльність**

Діяльність є окремим кроком у обчислювальному процесі. Одній діяльності відповідає окремий стан у системі і, принаймні, одна вихідна транзакція.

Діяльності можуть формувати ієрархічні структури, це означає, що діяльність може бути складено з декількох «менших» діяльностей, у цьому випадку вхідні і вихідні транзакції мають відповідати вхідним і вихідним транзакціям докладної діаграми.

**дія** (action) — деяка атомарна (яку не можна перервати зовні) операція, виконання якої призводить до зміни стану чи до повернення деякого значення

**діяльність** (activity) – неатомарна, з можливістю переривання, сукупність окремих обчислень, виконуваних кінцевим автоматом, які приводять до деякого результату або дії (action).

Кожна діаграма діяльності повинна мати один і тільки один початковий стан:

Рисунок - Початковий стан

Усі види діяльності (введення, виведення, розрахунки) слід зображати у вигляді овалу (слід відрізняти від еліпсу та прямокутника з закругленими кутами):

Рисунок - Діяльність

Перехід між окремими діяльностями завжди зображують у вигляді стрілки:

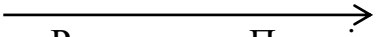


Рисунок — Перехід

Розгалуження, як і у блок-схемі, зображується ромбом. Але, на відміну від блок-схем, умови переходу записують не всередині ромбу, а біля стрілок у квадратних дужках:

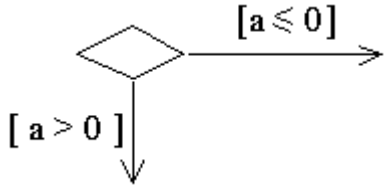



Рисунок - Розгалуження

Кожана діаграма повинна мати один або декілька кінцевих станів:

Рисунок - Кінцевий стан

На відміну від традиційних блок-схем, на діаграмі діяльності можна зобразити дії, які виконуються паралельно. Розгалуження на такі дії починається та закінчується лінійкою синхронізації.

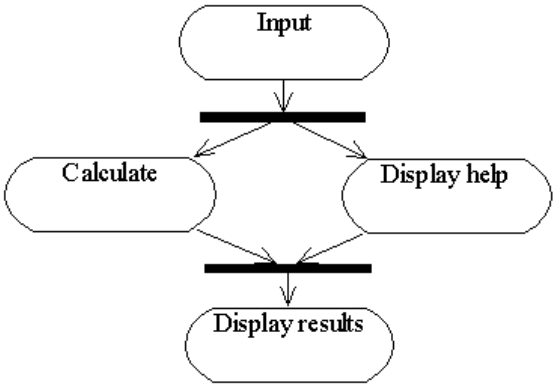



Рисунок – Паралельні гілки

Наступна діаграма представляє алгоритм методу дихотомії для розв'язання рівняння на інтервалі [a, b]:

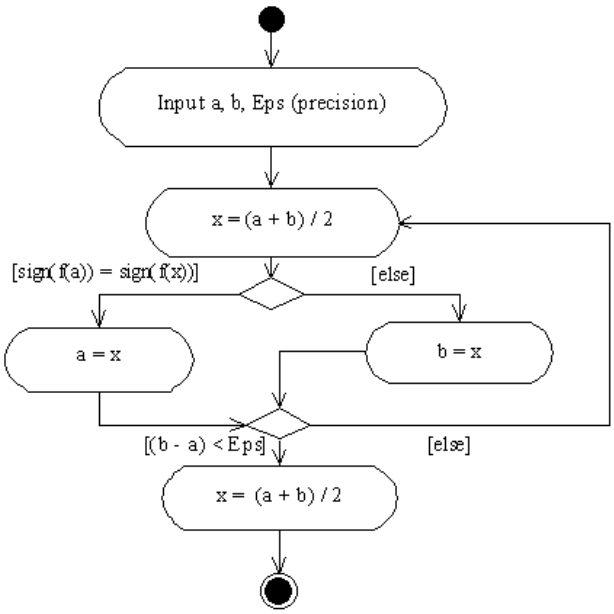



Рисунок - Метод дихотомії

Для того, щоб відобразити відповідальність окремих акторів (пакетів, компонентів, підсистем) за певні дії, вводиться поняття доріжки (swimlane). Вони зазвичай мають імена.

**Переваги блок-схем**:

— мають високу наочність подання алгоритму для нескладних алгоритмів.

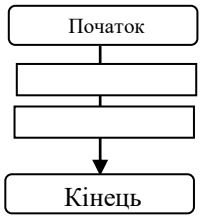
Недоліки:

— при зображенні великих та складних алгоритмів блок-схема розростається в довжину та ширину;

— на граф блок-схеми не накладаються обмеження, тому перетворення у еквівалентну програму, яка оформлена структурним стилем, необхідні відповідні перетворення.

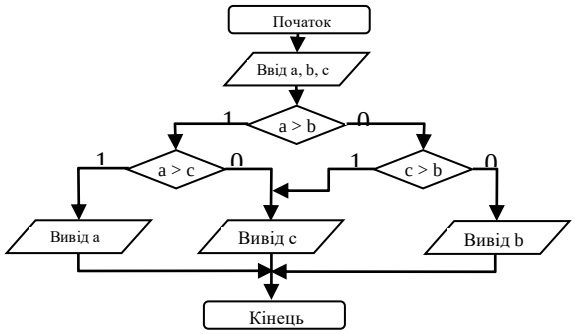
## Типи алгоритмів за структурою

1) Послідовний або лінійний



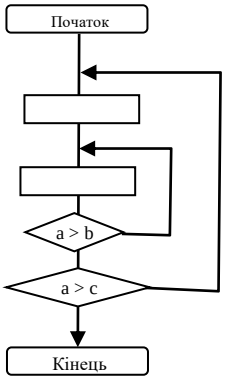

2) Розгалужений

Є розгалуження, але немає зворотних зв’язків



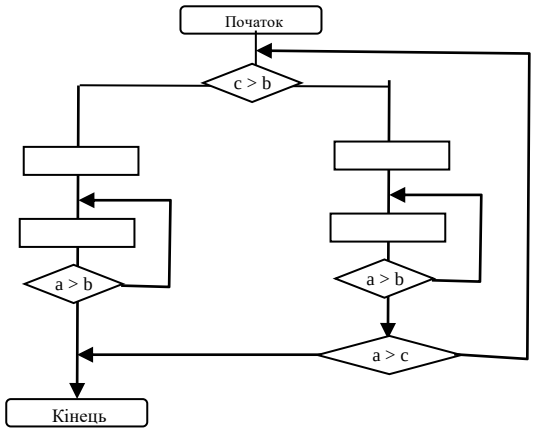

3) Циклічний

Є зворотні зв’язки, за якими фрагмент алгоритма повторює своє виконання



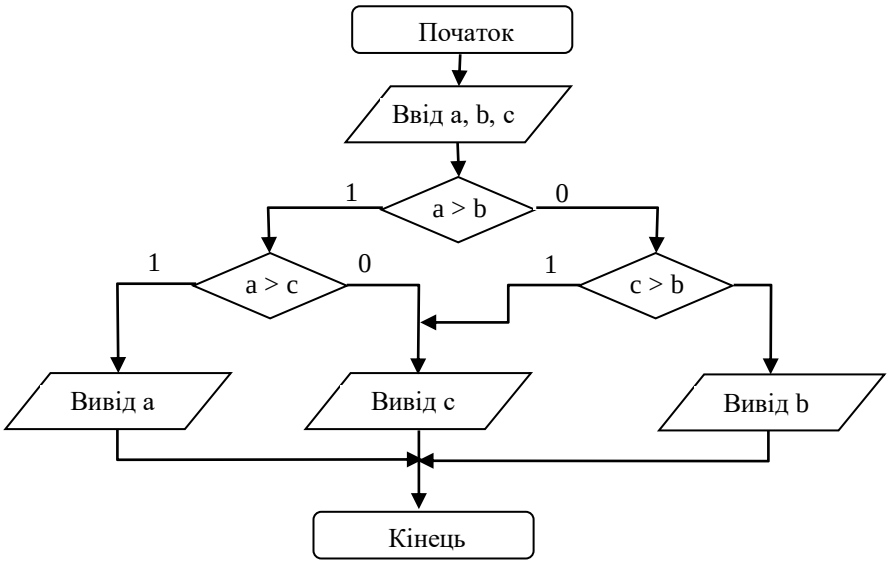
4) Складний (комбінований)

Є комбінацією 3-х попередніх типів алгоритмів




Приклад блок-схеми з умовними конструкціями

Задача — Знайти максимальне з трьох чисел a, b, c. Блок-схема:

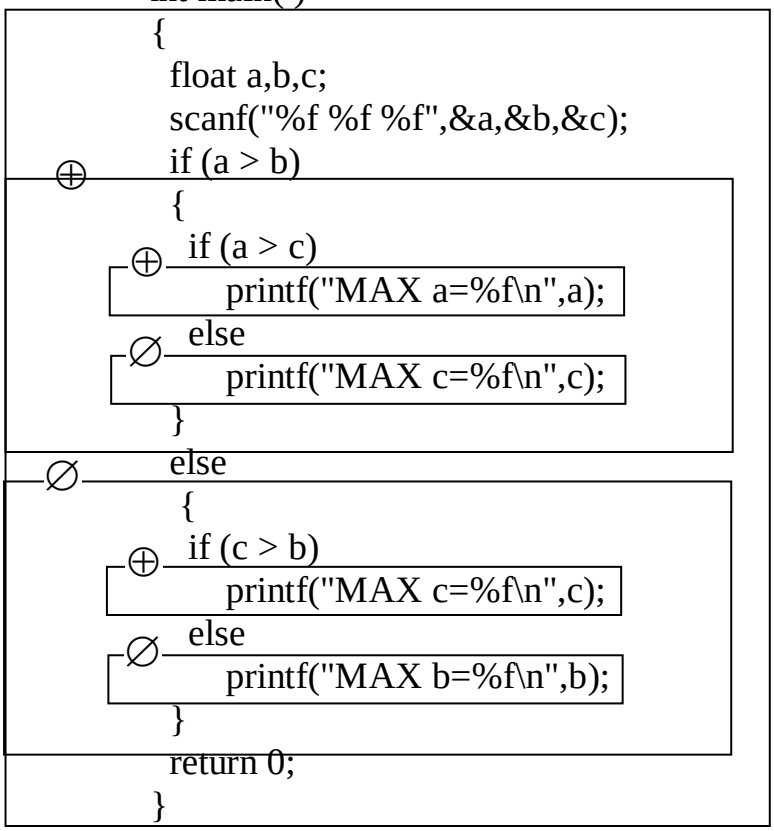



Опис мовою Бейсік

```
 10 INPUT A,B,C
 20 IF A>B THEN 30 ELSE 40
 30 IF A>C THEN 50 ELSE 70
 40 IF C>B THEN 70 ELSE 90
 50 PRINT “MAX A=”,A
 60 GOTO 100
 70 PRINT “MAX C=”,C
 80 GOTO 100
 90 PRINT “MAX B=”, B
100 END
```

Про структуру програми можна лише здогадуватись, проводячи лінії передачі керування.

На Сі структурним стилем:



# Лекція 7

## Основи структурного програмування

**Теорія схем програм** вивчає структури алгоритмів з мінімізованими лінійними ділянками безвідносно семантики їхніх операторів, тобто, їхні графи потоків керування.

В теорії схем програм було помічено, що деякі блок-схеми не піддаються аналізу. Тому було натурально виділити такий вид блок-схем, які легко аналізуються. Це вперше зробили італьянські вчені С. Бем і К. Джакопіні у 1966 г. А Егстер Дейкстра у 1968 р. опублікував статтю „A Case against the GO TO Statement" (Доведення проти використання оператора GO TO) яку редактор Н.Вірт виклав як „Go To Statement Considered_harmful" (Оператор GO TO вважається шкідливим).

Ця стаття викликала велику хвилю у світі і дала поштовх до зміни відношення до технології програмування, а саме, започаткувала появу парадигми структурованого (структурного) програмування.

Головна мета структурованого програмування — збільшення продуктивності програмістів, зменшення частоти помилок у програмах та спрощення сприйняття програм іншими програмістами.

**Структурний стиль** – це Єдність наступних складових програми

1) Структура інформаційного простору. ***Інформаційний простір*** задачі — повний набір об'єктів, з якими має справу обчислювальна задача.
2) Структура керування. Строге обмеження набору керуючих конструкцій: умовні, селективні та циклічні конструкції, усі гілки яких сходяться у одній точці програми; усі процедури вертають значення лише у точку їхнього виклику.
   ***Структурні переходи*** — переходи лише вперед або на більш високий рівень ієрархії і ніколи — за границі даного програмного модуля (власне, те, на що здатен оператор GOTO).
3) Потоки передачі даних. Задача розкладається на підзадачі, між якими дані передаються як потоки. Потоки виникають і споживаються у інтерфейсах підзадач. Заборонено передавати дані зсередини однієї задачі усередину іншої.
4) Структури даних. Логічно з'вязані дані об'єднуються у множини — структури даних, які відповідають структурі задачі.

Отже, у структурному стилі цілком можливе та часто виправдане використання структурного переходу GOTO.

При структурному програмуванні присвоювання та локальні дії стають органічною основною частиною програми.

При цьому кожна змінна служить лише для однієї мети. Недопустимо використовувати змінну для присвоєння іншого, невластивого їй значення з метою „економії" пам'яті. Це суттєво заплутує сприйняття програми і не дає ніякої справжньої економії.

**Основи оцінювання алгоритмів**

1. Загальна тривалість виконання алгоритму (***часова складність***)
2. Розмір алгоритму (***просторова складність***, загальна кількість комірок пам'яті для зберігання програми та вхідних і проміжних даних).

Часова складність оцінюється формулою

$$T = k_1 t_1 + k_2 t_2 + \ldots + k_n t_n = \sum_{i=1}^{n} k_i t_i .$$

де $t_i$ — тривалість виконання $i$-ї операції алгоритму
$k_i$ — кількість виконаних операцій $i$-го типу.

Для кожної обчислювальної моделі $t_i$ є специфічними значеннями.
Є **стандартні моделі** для оцінки часової складності. Це
— машина Т'юрінга,
— процесори MIX, MMIX, запропоновані Д.Кнутом,
— суперскалярний процесор DLX,
— VAX 11/780 (при виконанні тесту Dhrystone ця машина має продуктивність 1 DMIPS).

Для RISC-процесорів
однакова тривалість виконання більшості операцій алгоритму, для яких дані є у внутрішній регістровій пам'яті і дорівнює 1 такт.
Для операцій з даними в оперативній пам'яті ця тривалість зростає у кілька разів.
Для операцій переходу ця тривалість зростає на довжину конвеєра команд.

У реальних процесорах $t_i$ залежить від
—Тактової частоти
—наявності та розмірів кеш-пам'яті
—наявності та складності системи керування пам'яттю ( захисту пам'яті та віртуальної адресації)
—чи є процесор суперскалярним і кількості в ньому потоків команд
—наявності та ефективності механізму передбачення переходів, багатопотокового виконання.

Розрахунок $T$ важливий для систем реального часу (СРЧ).
СРЧ – обчислювальна система, час виконання алгоритму керування (реакції системи) у якій не повинен бути більшим за заздалегідь визначену границю.
Тому СРЧ будують на мікроконтролерах, у яких чітко визначені параметри $t_i$ . Для них розроблені методики вимірювання значення $T$.
В учбових цілях виділяють
$K_C$ — к-ть операцій порівняння
$K_A$ — к-ть арифметичних операцій
$K_L$ — к-ть повторень циклу

$K_{as}$ — к-ть операцій присвоювання

Причому $t_i$ = const. Це збігається з тим, що команди у сучасних процесорах в середньому, мають однаковий час виконання. Крім того, якщо тривалість операцій має однаковий порядок, то у оцінці складності найбільше значення має кількість операцій, а не їх тривалість.

Тривалість $T$ можна виразити поліномом

$T = a + bn + cn^2 + \ldots + en^m$

Тут $n$ — параметр, який виражає складність алгоритму. Це, наприклад, розрядність даних при обчисленні елем. Функції з заданою розрядністю, кількість вхідних даних, розмірність структури даних і т.п.

Два алгоритми є ***одного порядку складності***, якщо час $T$ для них можна виразити поліномом одного й того самого ступеня $m$. Цей факт виражається як

$T = O(n^m)$.

Наприклад, два алгоритми мають складності

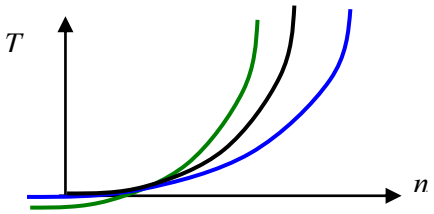
$T_1 = 3n^2 - 5n + 20$ і

$T_2 = 2n^2 + 7n - 5$.

Вони мають однаковий порядок складності $O(n^2)$.

В усякому разі, їхня складність виражається графіком залежності часу від параметра $n$, який показує, що складність алгоритму зростає квадратично, а не лінійно.

У таких випадках нехтують значеннями складових при молодших степенях змінної $n$.




## Приклад алгоритму розрахунку

$S = 1+2\cdot 2 + 3\cdot 3\cdot 3 + \ldots + n\cdot n\cdot n \ldots n = \sum_{i=1}^{n} i^i$.

Лінійний алгоритм розіб'ємо на ітерації, які виконуються гніздом циклів.

Індекс внутрішнього циклу j, зовнішнього — i.

Тоді обчислення виглядає як:

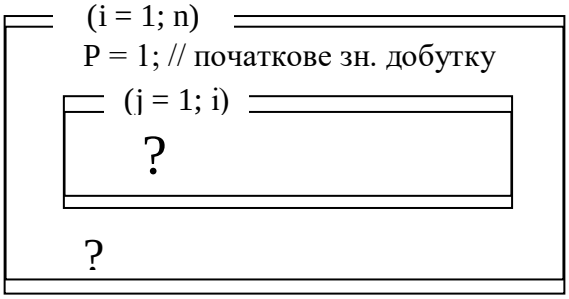
$$S = \left| \begin{matrix} i=1 \\ 1 \\ j=1 \end{matrix} + \left| \begin{matrix} i=2 \\ 2 \cdot 2 \\ j=1 \quad j=2 \end{matrix} + \left| \begin{matrix} i=3 \\ 3\cdot \quad 3\cdot \quad 3 \quad +\ldots \\ j=1 \quad j=2 \quad j=3 \end{matrix} \right| \left| \begin{matrix} i=n \\ n \cdot n \cdot \quad n \cdot \ldots \; n \\ j=1 \quad j=2 \; j=3 \quad j=n \end{matrix} \right| \right|$$

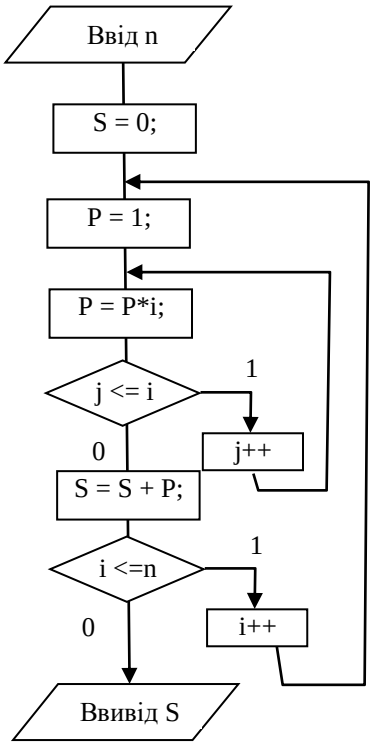
Структура програми виглядає як:

Scanf(“%d”,&n);

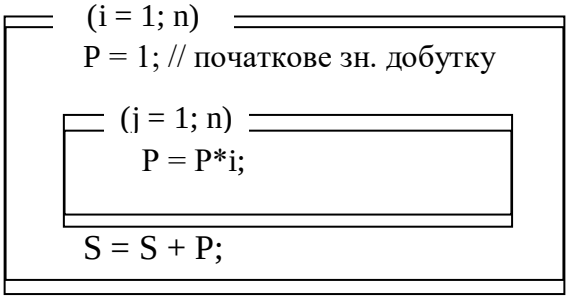
Int i, j ;

int S = 0; //початкове зн. суми




S = 0; //початкове зн. суми




```
for (i = 1; i <= n; i++)
{
  P = 1;
  for (j = 1; j <= i; i++)
        P = P*i;                // P *= i;
  S = S + P;                    // S += P;
}
```

**Розрахунок складності**

Кількість множень $k_1 = 1 + 2 + 3 + \ldots + n = \frac{(n+1)n}{2}$.

Кількість додавань $k_2 = n + n - 1 + \frac{(n-1)n}{2}$.

Кількість переходів $k_3 = n + \frac{(n+1)n}{2}$

Складність $\Theta = k_1 + k_2 + k_3 = 3n + \frac{(3n+1)n}{2} - 1 \approx 3n^2/2 = O(n^2)$.

# Лекція 8
# Найпростіші алгоритмічні прийоми

Приклад. Скласти програму розрахунку числа Пі

$$\pi/4 = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \ldots = \sum_{i=1}^{n} \frac{(-1)^{i+1}}{2i-1}.$$

Проблема: як керувати знаком суми.

**Алгоритм А1** — використання множника 1 з різним знаком.

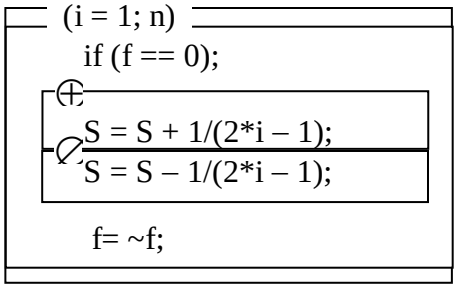
```
int  n, i;
float S;
int Z; // 0|1
scanf("%i",&n);
S = 0; //початкове зн. суми
Z = 1;
(i = 1; n)
  S = S + Z/(2*i – 1);
  Z = – Z;
printf("%r",S);
```

**Алгоритм А2** — перевірка парності номеру.

```
scanf("%i",&n);
S = 0; //початкове зн. суми
(i = 1; n)
  if (i%2 == 0);
    S = S – 1/(2*i – 1);
    S = S + 1/(2*i – 1);
printf("%r",S);
```

**Алгоритм А3** — використання булевської змінної.

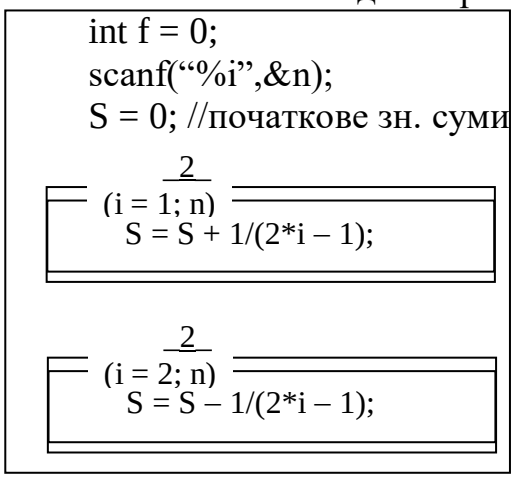
```
int f = 0;
scanf("%i",&n);
S = 0; //початкове зн. суми
```




```
printf("%r",S);
```

**Алгоритм А4** — виконання двох проходів.

```
int f = 0;
scanf("%i",&n);
S = 0; //початкове зн. суми
```




```
printf("%r",S);
```

**Метод динамічного програмування**

— в теорії обчислень — спосіб вирішення складних задач через розкладання їх на простіші підзадачі.

Програмування "динамічне" через те, що структура алгоритму змінюється динамічно в залежності від етапу алгоритму або значення даних.

Під час динамічного програмування часто виконується мемоізація.

**Мемоізація** (англ. memoization) — спосіб зменшення обсягів обчислень за рахунок збереження результатів обчислення функцій та невиконання повторних обчислень.

**Алгоритм А5** — заміна множення на додавання.

```
scanf("%i",&n);
float D = 1;
Z = 1;
S = 0: //початкове зн. суми
(i = 1; n)
    S = S – Z/D;
    D = – D – 2;
    Z = – Z;
printf("%r",S);
```

Тут не обчислюється знаменник як добуток, а береться збережене значення з попередньої ітерації і додається інкремент.

Задача. Розрахувати

$$S = 1+1\cdot 2 + 1\cdot 2\cdot 3 +\ldots+ 1\cdot 2\cdot 3\ldots n = \sum_{i=1}^{n} j!.$$

Тоді обчислення виглядає як:

| S = | i=1 | i=2 | i=3 | i=*n* |
|---|---|---|---|---|
| | 1 | 1 * 2 | 1· * 2· * 3 | 1 * 2 * 3 *· ... |
| | | + | +… | *n* |
| | j=1 | j=1 j=2 | j=1 j=2 j=3 | j=1 j=2 j=3 |
| | | | | j=*n* |

Програма з гніздом циклів

```
S = 0; //початкове зн. Суми
(i = 1; n)
    P = 1; // початкове зн. добутку
    (j = 1; i)
        P = P*j;
    S = S + P;
```

програма з мемоізацією

```
S = 0;
P = 1; // початкове зн. добутку
(i = 1; n)
    P = P*i;
    S = S + P;
```

# Лекція 8
# Підпрограми

**Підпрограма** — це група операторів програми, яка зберігається окремо від місця, де вона потребується для виконання і має унікальне ім'я, за яким вона викликається для виконання.

**Опис підпрограми** — завдання сукупності дій, які позначаються як підпрограма.

**Виклик підпрограми** — застосування її.

Спочатку підпрограми використовувались як засіб зменшення опису алгоритму через "винесення за дужки" його загальних частин.

Потім поняття підпрограми стало розумітись як засіб формування бібліотек. І тоді **бібліотечна підпрограма** стала використовуватись у багатьох програмах, в тому числі програмах різних розробників.

Далі, коли програму почали розділяти на модулі, роль підпрограм зросла — вони стали основою модулярізації.

**Модулярізація —** це розділення програми на відносно незалежні частини, між якими вказані явно взаємозалежності, які можна незалежно розробляти і використовувати.

Підпрограми — це основний засіб абстрагування у програмуванні.

**Іменування** — позначення підпрограми іменем, яке заміщує її алгоритм виконання при її використанні.

При цьому при розробці підпрограми програміст *абстрагується* від рівня її виконання — не знає де і як вона буде виконана.

При використанні підпрограми програміст не знає як вона запрограмована і як виглядає всередині — він розглядає її як єдине ціле — *абстрактне*.

Існують два види підпрограм:

— **функції**, виконання яких породжує значення, яке передається в точку виклику; виклик функції є елементом виразу;

— **процедури**, виконання яких використовується лише для перетворення даних, які є доступними у контексті виклику і не породжує значень; виклик процедури – це оператор.

У мові Сі ці два поняття об'єднані за допомогою спеціального значення ***нічого,*** тобто, **void**. Тоді процедура — це функція, яка виробляє значення void.

Бібліотека Сі — сукупність різних описів, в тому числі описів функцій. Для її використання необхідне підключення до програми відповідного заготовочного файла за допомогою оператора #include.

У мові Сі засоби модулярізації програм пов'язані з поняттям заголовочного файла – header file – в якому явно описується що представляється модулем, який має такий заголовочний файл. Це по суті — інтерфейси підпрограм.

**Контекст**— сукупність імен об’єктів (змінних), над якими виконуються дії у даному програмному фрагменті.

**Локальний контекст** — контекст, який не поширюється далі програмного фрагменту і є невидимим ззовні його.

**Глобальний контекст** — контекст, який використовується як в даному програмному фрагменті, так і в його оточенні.

**Інтерфейс підпрограми** — сукупність описів глобального контексту (**параметрів**), який явно передається у підпрограму (контекст виклику) та з неї.

**Реалізація підпрограми** — сукупність описів з використанням локального контексту, який не передається у фрагмент програми, що її викликає.

**Блок** — тіло підпрограми разом з її локальним контекстом.

В Сі іменування задається заголовком функції з обов’язковим її ім’ям. Наприклад:

```
const NN = 100;
typedef float Matrix[NN][NN];

void MyProc(Matrix M, const int SizeM)
int MyFunc(NN)
```

Тут MyProc, MyFunc — імена функцій. Те, що це процедура — вказується специфікатором void.

Тут глобальний контекст —

– константа NN,

– змінна SizeM,

– змінна M типу Matrix,

– змінна, яка повертається функцією MyFunc.

Тіло підпрограми — алгоритм її реалізації — описується слідом за заголовком у операторних дужках { і }:

```
int MyFunc(NN)   // заголовок
{
int N;           // опис локального контексту
  N = NN*NN*8;   //  тіло функції
  return N;
}
```

Тут оператор return N показує, яким чином значення функції передається у точку її виклику.

Слід запам’ятати, що в мові Сі не можна визначати одну функцію всередині іншої.

## Виклик підпрограми

Синтаксично виклик підпрограми оформлюється як використання в тексті програми імені процедури чи функції. Процес, який запускається під час передачі керування підпрограмі, розбивається на 3 фази:

У першій фазі архітектура мови (Підготовка виконання підпрограми):

1. Формує локальний контекст — забезпечує можливість оперування зі змінними, які доступні для операторів тіла підпрограми.
2. Виконує вхід в підпрограму — передає керування на тіло підпрограми з запам'ятовуванням точки повернення.

У другій фазі: (Виконання алгоритму підпрограми):

3. виконуються оператори тіла підпрограми.

У третій фазі (Закінчення виконання підпрограми):

4. В архітектурі знищується локальний контекст підпрограми і вивільняється пам'ять.

5. Передається керування у точку повернення.

При виконанні програми, під час виклику підпрограми з певним ім'ям, архітектура комп'ютера повинна знайти це ім'я у сховищі імен підпрограм, яке називається **таблицею імен.** У цій таблиці, крім ім'я підпрограми, вказується контекст, до якого це ім'я відноситься.

Якщо таке ім'я не знайдене, то це кваліфікується як помилка конструкції виклику.

## Способи передачі параметрів у підпрограму

**1. Параметризація** — коли частина локальних даних підпрограми, що називається сукупністю **формальних параметрів**, об'являється виділеною для передачі даних між локальним контекстом підпрограми та контекстом її виклику, тобто, сукупністю **фактичних параметрів**.

**2. Передача через загальні блоки —** деякий зовнішній контекст передається для використання як внутрішній контекст. Причому цей контекст є доступним для використання кількома підпрограмам.

**Узгодженність** типів фактичних параметрів типам формальних параметрів — властивість параметризації, коли кожен фактичний параметр повинен бути таким, який потребує алгоритм процедури.

При параметризації використовується:

**Приведення значень типів** — правила перетворення значень одного типу до значень іншого. Для виконання приведення

— або підбирається підпрограма з такою самою назвою але з відповідним типом параметрів (псевдонім – alias),

— або неявно викликається підпрограма перетворення типів.

У першому випадку говорять про **поліморфізм процедур.**

1) **Передача параметра за значенням**, коли перед виконанням алгоритму підпрограми на фазі її підготовки для такого параметра в локальному контексті виділяється спеціальна змінна — ідентифікатор формального параметру, яка під час виклику одержує значення фактичного параметра.

```
void P ( int x )
{
a = x; x = 1;
}
```

2) **Передача параметра як змінної**, при якій фактичний параметр є змінною, яка заміняє локальну змінну в тілі підпрограми у всіх випадках її використання у фазі виконання алгоритму підпрограми.

В мові Сі це виконується за допомогою вказівника на змінну.

3) **Передача параметра за посиланням**, коли при виконанні виклику підпрограми на фазі виконання формальний параметр заміняється на адресу фактичного параметру.

```
void S(int *x)
```

— вказує, що фактичним параметром буде щось, яке вказує на ціле число. При цьому цей вказівник можна добути операцією взяття адреси

```
 & variable
```

під час виклику.

**Константні параметри** — параметри, які виділені як такі, що не можуть бути змінені під час виклику підпрограми.

```
void P(const int x);
```

**Як відбувається насправді виклик і передача параметрів в зкомпільованій програмі на Сі.**

Через стек.

Лекція 9

# Структури даних

**Структура даних** – програмний об’єкт для зберігання та обробки множини однотипових та/або логічно зв’язаних даних (складеного конструктивного об’єкта).

Чому структура?

Це множина даних, внутрішня будова якої формується за деяким законом. Тобто, такі дані мають свою структуру.

Структура даних формується за допомогою типу даних, посилання та обробляється за допомогою відповідних операцій та функцій, які притаманні даній мові програмування.

Види структур даних:

— масив
— структура (запис)
— об’єднання (union)
— список
— дерево
— хеш-таблиця.

**Абстрактний тип даних** (abstract data type) – це визначуваний тип, що потребує опису не тільки множини значень, але й множини операцій.

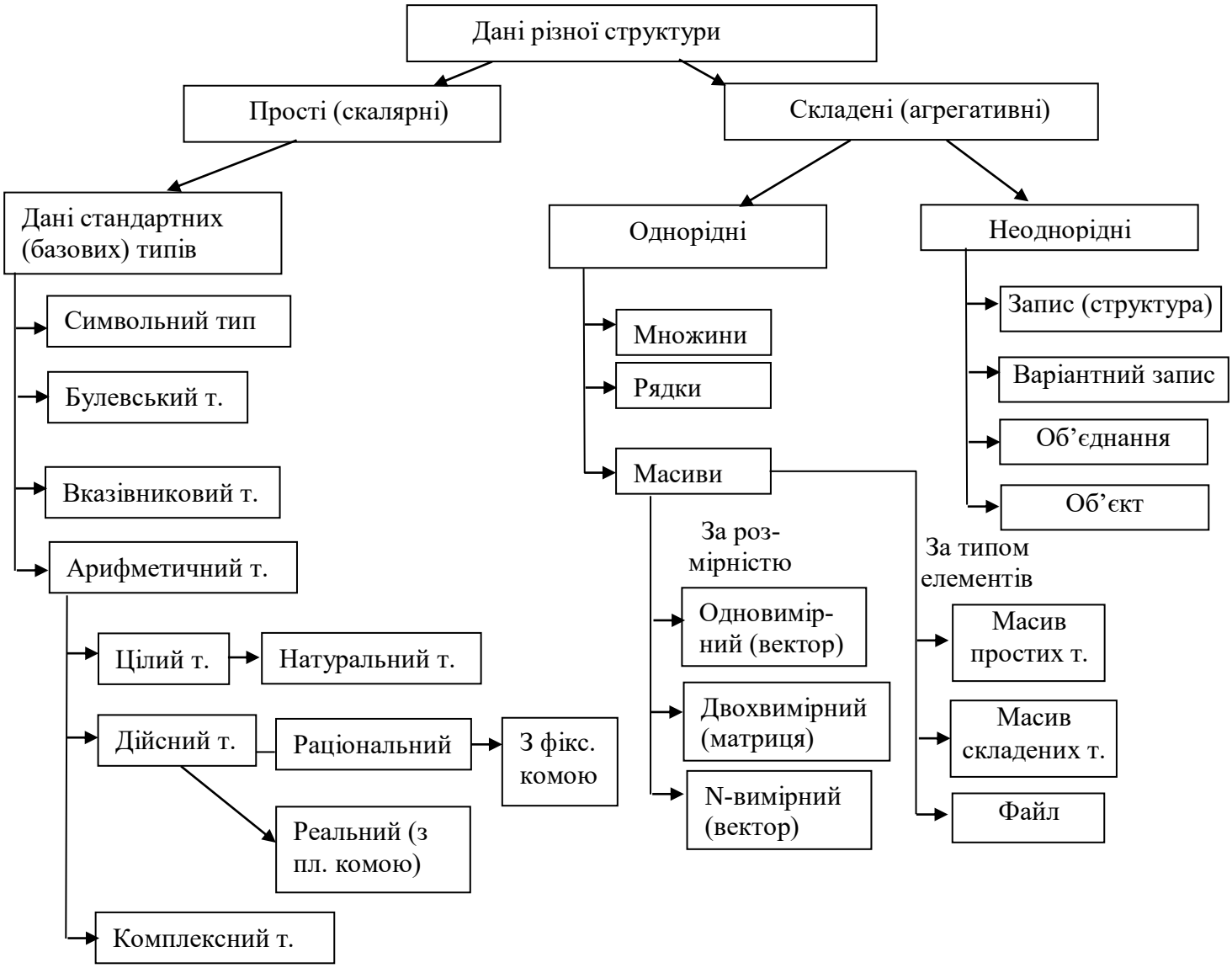
Приклади: список, дерево, черга.

**Функції над структурами даних**

**конструктори** - змінюють стан об'єкта (наприклад, записати в, прочитати з);
**селектори** - оцінюють стан об'єкту (наприклад, чи порожнє значення структури даних, або повне; яка довжина значення, що в голові значення, а, що у хвості, якщо це список; якщо, наприклад, це черга - що у вершині?
**ітератори** – «розглядають» (досліджують) стан об'єкта (наприклад, повертають, значення всіх компонентів послідовне одне за одним без зміни стану об'єкта).

| Дані | |
|---|---|
| ***Дані статичної структури***<br>— дані, взаєморозташування та взаємозв’язки яких є постійними незалежно від алгоритму та вхідних даних задачі. | ***Дані динамічної структури***<br>— дані, внутрішня будова яких формується за деяким законом, але кількість елементів, їх взаєморозташування та взаємозв’язки динамічно змінюються під час виконання алгоритму, не порушуючи закону формування цієї структури даних. |

Розглянемо класифікацію типів даних з метою визначення відношення структур даних до інших способів їх представлення




***Однорідна структура даних*** — структура, усі елементи якої мають однаковий тип

***Неоднорідна структура даних*** — структура, яка може об'єднувати в єдине ціле дані різних типів

# Масиви

**Масив** — це структура даних, яка представляє однорідну, фіксовану за розміром і конфігурацією сукупність елементів простої або складеної структури, причому елементи масиву упорядковані за номерами.

Масив визначається іменем (ідентифікатором), який є загальним для усіх його елементів, а також кількістю елементів і розмірністю, які необхідні для місцезнаходження конкретного елемента.

До елементів масиву звертаються одного або декількох індексів (в залежності від розмірності масиву).

**Одномірний масив**

Декларується як:

int arr[n];

Результат декларації — призначення 0-му елементу масиву конкретної символічної адреси arr та виділення n комірок пам'яті для зберігання чисел типу int. Після декларування на місці масиву зберігаються випадкові дані.

У більшості мов програмування одномірний масив заповнюється елемент за елементом найчастіше за допомогою оператора циклу.

Хоча у діаграмі дій одним оператором можна переприсвоїти один масив іншому, у мові програмування це виконується поелементно у циклі.

У мові Сі масив завжди починається з 0-го елемента і має розміри *nk* байт, де *k* — розмір одного елемента масиву.

Розмір *k* елемента можна визначити за допомогою функції int sizeof(тип).

Наприклад, sizeof(int) може бути і 2, і 4 в залежності від компілятора.

Наприклад:

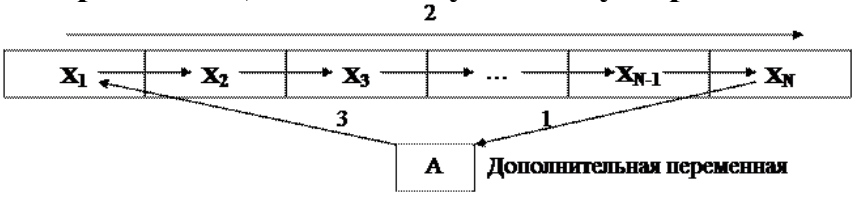
```
int a[10] = {1,2,3,4,5,6,7,8,9,0}; // масив ініціалізований при декларації
int b;
a[0] = –2;
for(i=1; i<9; i=i+1)    a[i]= a[i-1]+2;   //запис 9 елементів масиву
```

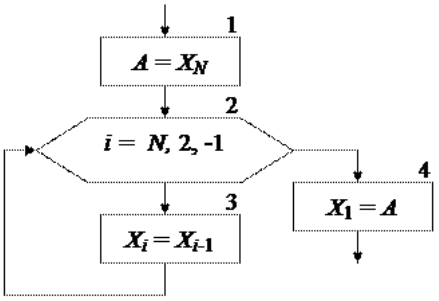
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| a = | -2 | 0 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 0 |

```
b = a[5]; //b = 8
```

# Алгоритми зсуву масиву

## Алгоритм #1 Циклічний зсув масиву вправо на 1 поз.




A= X(n)

–1

(i = n; 2)

X(i) = X(i–1);

X(1) = A

На 1-му етапі значення останнього елементу $X_N$ заноситься у додаткову змінну $A$ (блок 1).

На 2-му етапі виконується цикл FOR (блок 2), який перебирає елементи массива X у зворотному порядку, тобто з правого края (шаг -1), починаючи з $N$-го і кінчаючи 2-м елементом.

На 3-му етапі значення додаткової змінної $A$ буде занесено в перший елемент масива $X_1$.

Слід звернути увагу на граничні значення параметра цикла $i$, що змінюється від $N$ до 2. При підстановці них не повинне виконуватись звертання до неіснуючих елементів масиву.

Зсув вліво:

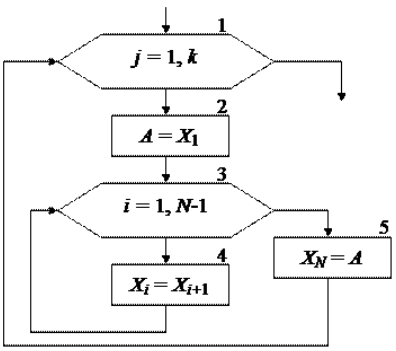
A= X(n)

+1

(i = 1; n–1)

X(i) = X(i+1);

X(n) = A

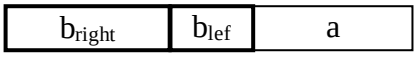



Зсув вліво на *k* позицій

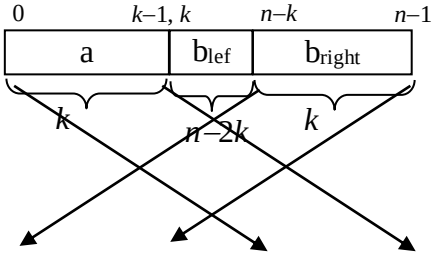
**Алгоритм # 2: перестановка блоків**

Можна запропонувати і інший алгоритм, який виникає з розгляду завдання під іншим кутом зору. Циклічний зсув масиву x зводиться фактично до заміни ab на ba, де a - перші *k* елементів x, a b — решта елементів.

Припустимо, що a коротше b. Розіб'ємо b на bleft і bright, де bright містить *k* елементів (стільки ж, скільки і a).

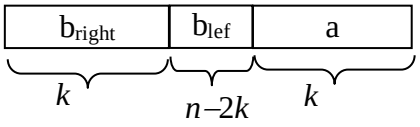



123 | 45 | 678




```
+1
(i = 0;k–1)
  T = A(i+n–k);
  A(i+n–k) = A(i);
  A(i)  = T:
```

Поміняєм місцями a і $b_{right}$, одержим $b_{right}b_{left}a$.



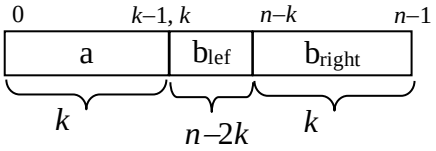

678| 45 |123

При цьому **a** з’явиться у кінці масиву — там, где і треба. Тому можна продовжити з перестановкою $b_{right}$ і $b_{left}$.

45| 678 | 123

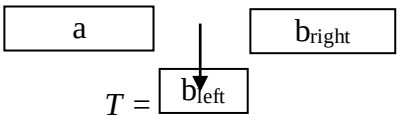
при цьому алгоритм застосовується рекурсивно.

**Алгоритм №3 зі збереженням середніх комірок масиву**



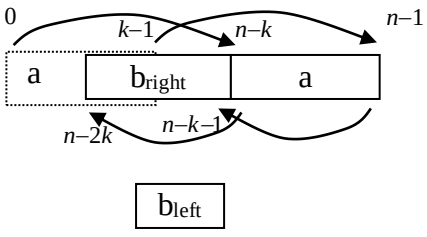

123 | 45 | 678

1 крок



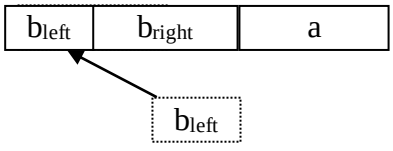

123 | | 678 45

2 крок




678 | 123 45

3 крок




45|678 | 123

A(n)

```
+1
(i = k; n–k–1)
  T(i–k) = A(i);

–1
(i = k–1; 0)
  A(i+n–2k) = A(i+n–k);
  A(i+n–k)  = A(i);

+1
(i = 0; n–2k–1)
  A(i) = T(i);
```

**Алгоритм № 4: переворотами**

Припустимо, що у нас є функція reverse, яка переставляє елементи деякої частини масиву в протилежному порядку. У початковому стані масив має вигляд ab. Викликавши цю функцію для першої частини, отримаємо $a^rb$. Потім викличемо її для другої частини: отримаємо $a^rb^r$. Потім викличемо функцію для всього масиву, що дасть $(a^rb^r)^r$, а це в точності відповідає ba.

/* *abc|defgh* */

1. reverse(0, i-1) /* *cba|defgh* */
2. reverse(i, n-1) /* *cba|hgfed* */
3. reverse(0, n-1) /* *defgh|abc* */

```
void reverse (int* a, int n);
```

яка змінює порядок елементів масиву на протилежний (останній елемент міняється місцями з першим, передостанній з другим і т.д). Її реалізація:

```
void reverse (int* a, int n) { //адреса масиву і довжина
   int i = 0;
   int j = n-1;
   while (i < j) {
      int t = a[i];
      a[i] = a[j];
     a[j] = t;
      ++i; --j;
   }
}
```

Для циклічного зсуву елементів масиву на k позицій вправо досить спочатку інвертувати початковий відрізок масиву довжини n-k, потім кінцевий відрізок довжини k і після цього інвертувати весь масив. Нижче наведено приклад для масиву довжини n = 10, що містить числа від 1 до 10, і величини зсуву k=3.

Вхідний масив:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10.

Спочатку інвертуємо початок довжини 7:

**7, 6, 5, 4, 3, 2, 1,** 8, 9, 10.

Потім інвертуєм кінець довжини 3:

**7, 6, 5, 4, 3, 2, 1,** *10, 9, 8*.

На третьому кроці інвертуєм весь масив:

*8, 9, 10,* **1, 2, 3, 4, 5, 6, 7**.

Ми отримали циклічний зсув елементів на 3 позиції вправо.

Отже, запишемо на Сі реалізацію функції циклічного зсуву елементів масиву на k позицій вправо за допомогою допоміжної функції інвертування масиву:

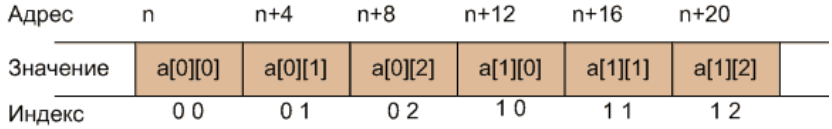
```
void shiftk(int *a, int n, int k) { //0 <= k < n
   reverse (a, n-k);   // Инвертируем начало массива длины n-k
   reverse (a+n-k, k); // Инвертируем конец массива длины k
   reverse (a, n);     // Инвертируем весь массив
}
```

# Лекція 10
# Двовимірні масиви

Загальна форма об’яви багатовимірного масиву
тип ім’я[розмір1][розмір2]...[розмір m];

Елементи багатовимірного масиву розміщені у послідовних комірках оперативної пам’яті за зростанням адрес. В пам’яті комп’ютера елементи багатовимірного масиву розміщені підряд, наприклад, масив, який має 2 рядка і 3 стовпця,
int a[2][3];

буде розміщений у пам’яті наступним чином




Загальна кількість елементів у двохвимірному масиві визначається як

**К-ть рядків * К-ть стовпців = 2 * 3 = 6.**

Кількість байт памяти, яка необхідна для розміщення масива, визначається як

**К-ть елементів * РозмірЕлементу = 6 * 4 = 24 байта.**

**Ініціалізація багатовимірного масиву**

Значення елементів багатовимірного масиву, можуть бути задані константами при об’яві, які обмежені фігурними дужками {}. Але вказівка кількості елементів в рядках і стовпцях має бути вказана в квадратних дужках [].

Приклад на Сі

```
#include <stdio.h>
int main()
{
  int a[2][3] = { 1, 2, 3, 4, 5, 6 };
  printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);
  printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);
  getchar();
  return 0;
}
```

```
#include <stdio.h>
int main()
{
  int a[2][3]; // массив из 2 строк и 3 столбцов
  int i, j;
  // Ввод элементов массива
  for (i = 0; i<2; i++)  // цикл по строкам
  {
    for (j = 0; j<3; j++) // цикл по столбцам
    {
```

```
      printf("a[%d][%d] = ", i, j);
      scanf("%d", &a[i][j]);
    }
  }
  // Вывод элементов массива
  for (i = 0; i<2; i++)  // цикл по строкам
  {
    for (j = 0; j<3; j++) // цикл по столбцам
    {
      printf("%d ", a[i][j]);
    }
    printf("\n"); // перевод на новую строку
  }
  getchar(); getchar();
  return 0;
}
```

## Робота з дисплеєм консолі

Бібліотека windows використовується для розробки застосунків Windows. Там задекларована велика кількість типів даних і констант, які заміщують собою стандартні типи данних мови Сі.

Така заміна дає змогу відокремити програмний інтерфейс Windows від самої операційної системи Windows та від конкретних реалізацій компіляторів мови Сі. Наприклад, в файлі windows.h визначено тип UINT, який замінює тип unsigned int :

```
typedef unsigned int UINT;
```

Цей тип для старих компіляторів та версій Windows дає 16-розрядні числа, а для нових — 32-розрядні.

Для наших потреб — вивчення алгоритмів поводження з матрицями — необхідні деякі константи, типи і функції для роботи з консоллю. Для підключення бібліотеки слід додати прагму:

```
#include <windows.h>
```

Розмір стандартного вікна консолі 80x25 = 2000 символів.

Для початку роботи з консоллю необхідно програмі одержати її дескриптор — адресу сегмента фізичної пам’яті консолі з правами доступу. Функція одержання дескриптора стандартного пристрою вводу-виводу чи помилки підключення має такий інтерфейс:

```
HANDLE WINAPI GetStdHandle(__in DWORD nStdHandle);
```

Тут константа nStdHandle може прийматиь значення

```
STD OUTPUT_HANDLE = -11;  // пристрій виводу
STD_ERROR__INPUT_HANDLE = -10;  // пристрій вводу
STD_ HANDLE = -12;  // помилка
```

Наприклад, для виводу на консоль :

```
HANDLE hout = GetStdHandle(STD_OUTPUT_HANDLE);
```

Тут hout — змінна, яка задає вікно, що програмується.

Для вказівки кодової сторінки використовується функція

```
BOOL WINAPI SetConsoleOutputCP(UINT wCodePageID);
```

Наприклад, для вказівки українського кодування:

```
SetConsoleOutputCP(1251);
```

Функція SetConsoleTextAttribute встановлює атрибути кольору текста (колір букв і фону місць з буквами). Вона використовує синтаксис:

```
BOOL SetConsoleTextAttribute(HANDLE,WORD);
```

і приймає два аргумента — дескриптор вікна (консолі) та число, біти якого визначають кольори. Усі символи, які виводяться після виклику цієї функції будуть мати встановлені атрибути.

Кольори задаються у молодшому байті слова WORD з бітами:

```
Ib Rb Gb Bb If Rf Gf Bf
```

Тут біти з індексом b задають колір фону (back), а з індексом f — колір символу (forward). Інтенсивність кольору (яскравість) задається бітами Ib та If . Наявність червоної, зеленої та синьої складових задається бітами Rb, Gb, Bb, Rf, Gf, Bf.

Координати точки у консольному вікні задаються структурою типу COORD. Наприклад, символ у 5-й позиції 2-го рядка має координати змінної Pos:

```
COORD Pos;
Pos.X = 5;
Pos.Y = 2;  // або можна COORD Pos = {5,2};
```

Функція SetConsoleCursorPosition встановлює положення текстового курсора в консольному вікні. Наприклад,

```
SetConsoleCursorPosition(hout, Pos);
```

Перший аргумент — дескриптор консольного вікна, а другий – змінна типа COORD, в якій знаходяться координати текстового курсора.

```
FillConsoleOutputAttribute(hout, 0, 2000, point, &l); // очистка экрана
```

Суть виконання лабораторної роботи №4 — виводити елементи матриці на консоль у заданому порядку динамічно, щоб мати змогу відстежити цей порядок. Для цього слід додати затримку виводу наступного елементу. Для цього можна використати наступну функцію:

```
VOID WINAPI Sleep(DWORD dwMilliseconds);
```

Яка зупиняє виконання даного програмного потоку на кількість мілісекунд, яка дорівнює параметру. Наприклад,

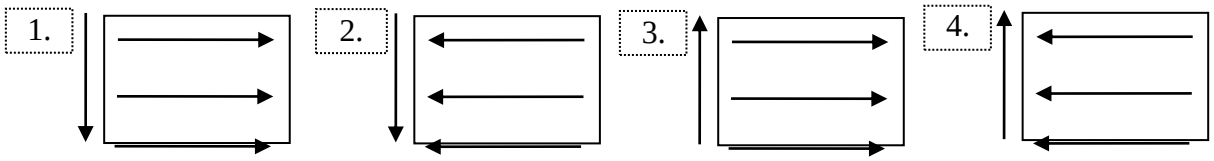
```
Sleep(200); // затримка на 0.2 секунди
```

Лекція 11

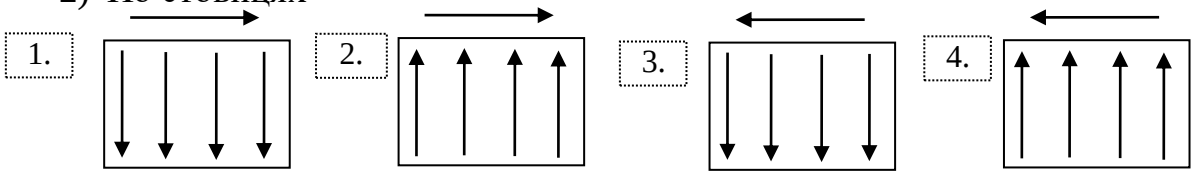
# Алгоритми обходу двохвимірного масиву

**Обхід (сканування)** матриці — це взяття кожного її елементу рівно один раз у довільному порядку.

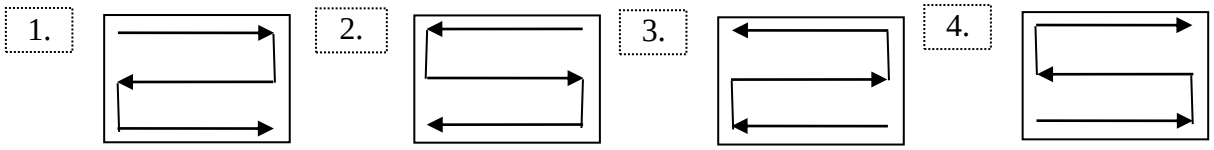
1) По рядках



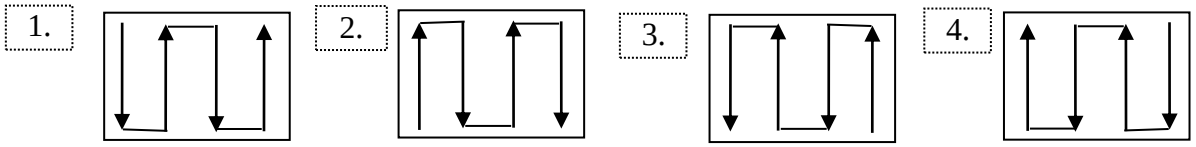

2) По стовпцях



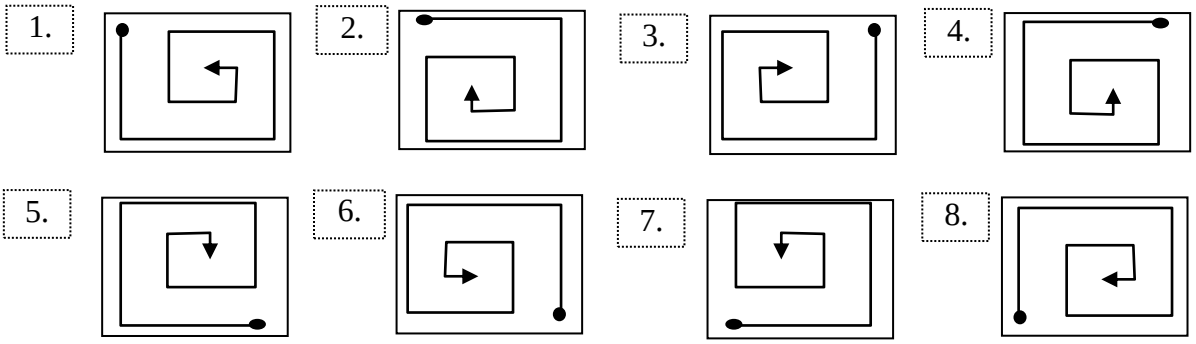

3) Змійкою по рядках



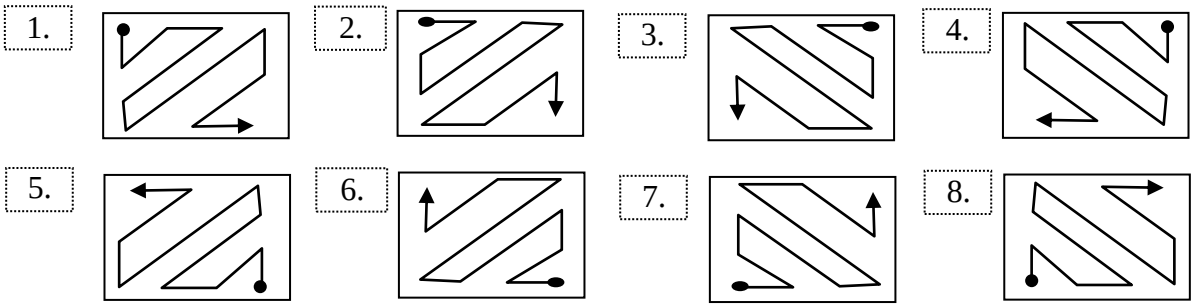

4) Змійкою по стовпцях



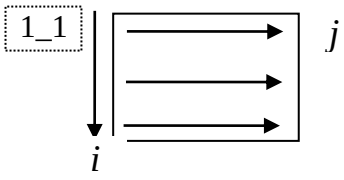

5) По спіралях

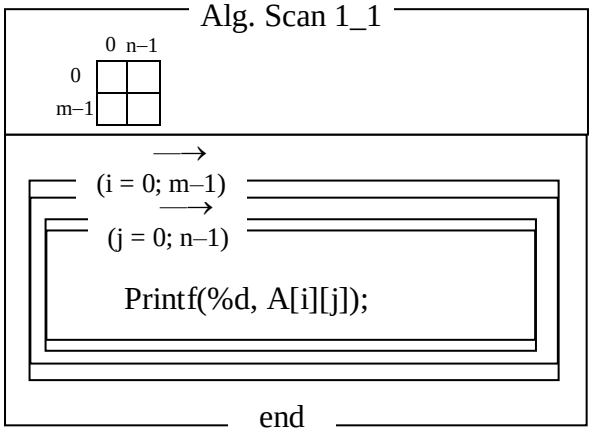


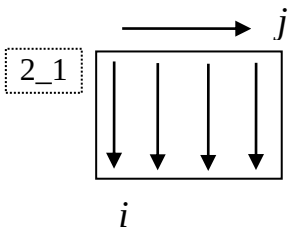
6) Змійкою по діагоналях

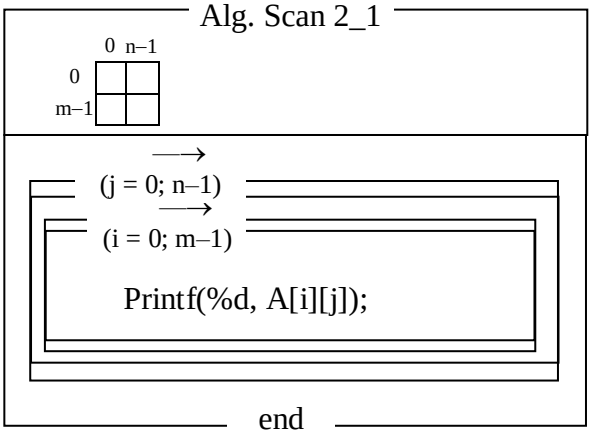


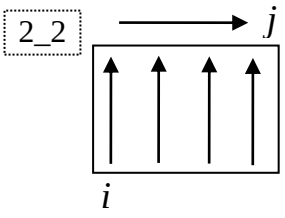

**Алгоритм обходу по рядках**

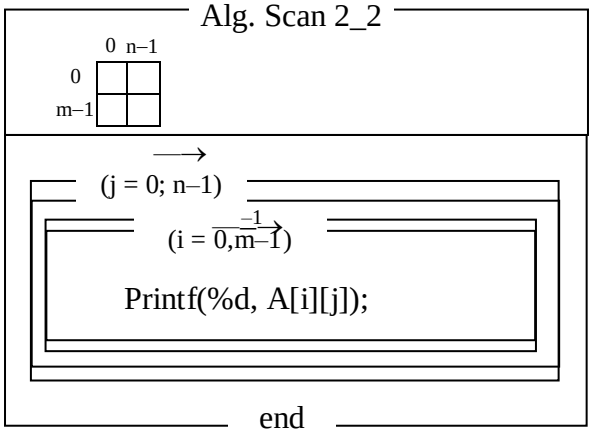


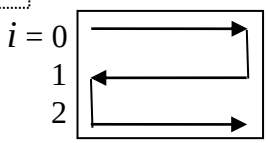



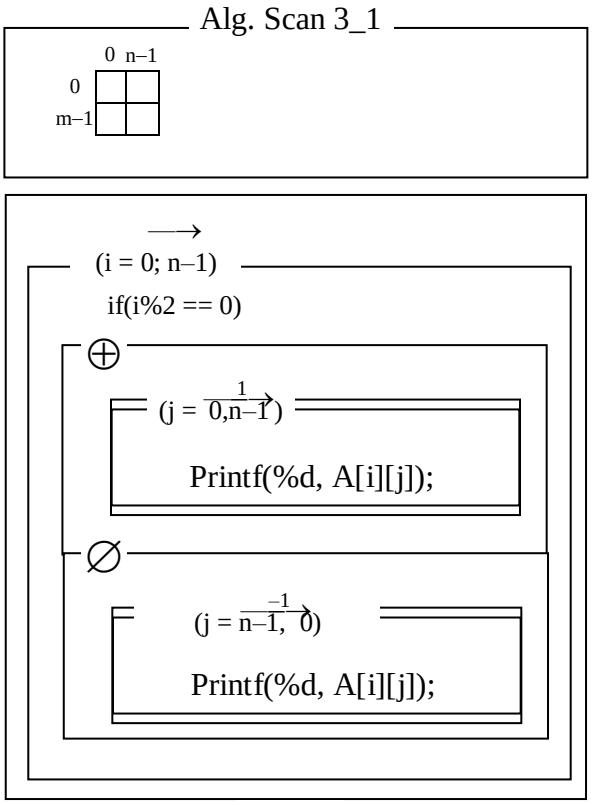



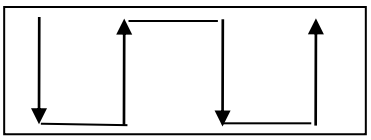
3_1



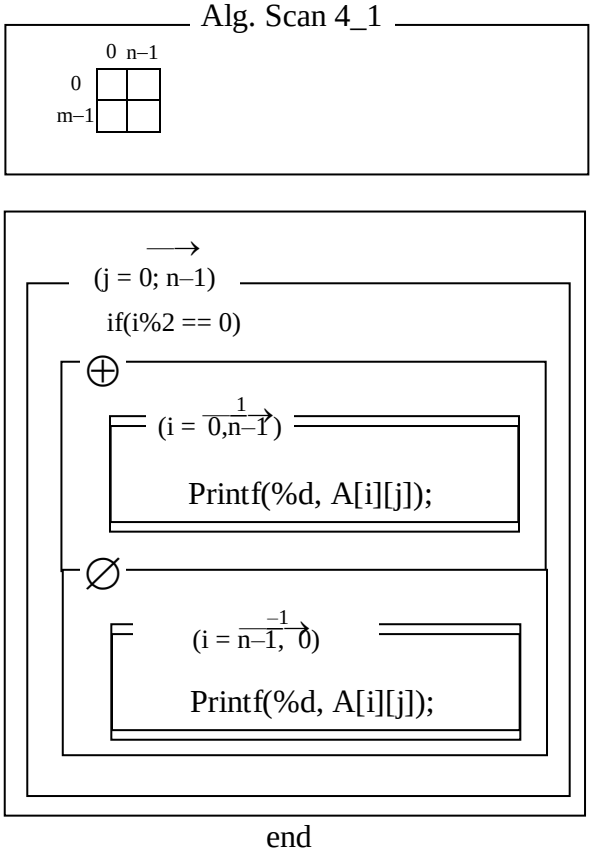




## 4. Змійкою по стовпцях

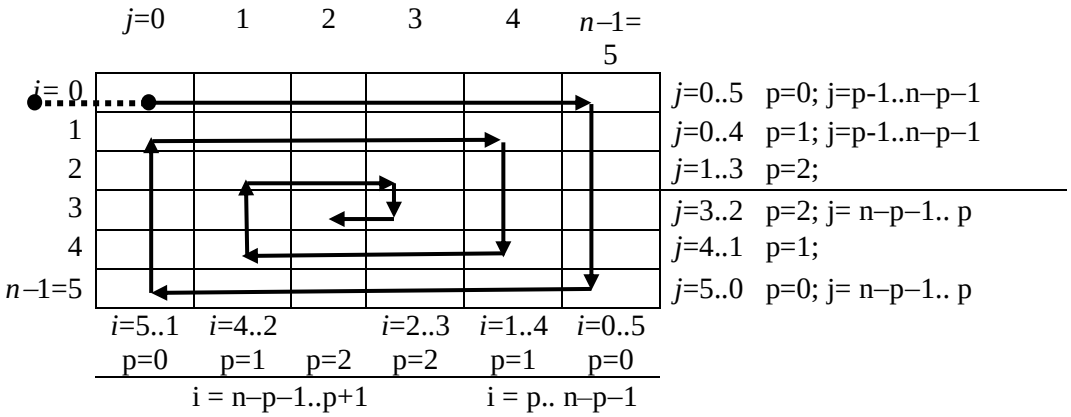
4.1



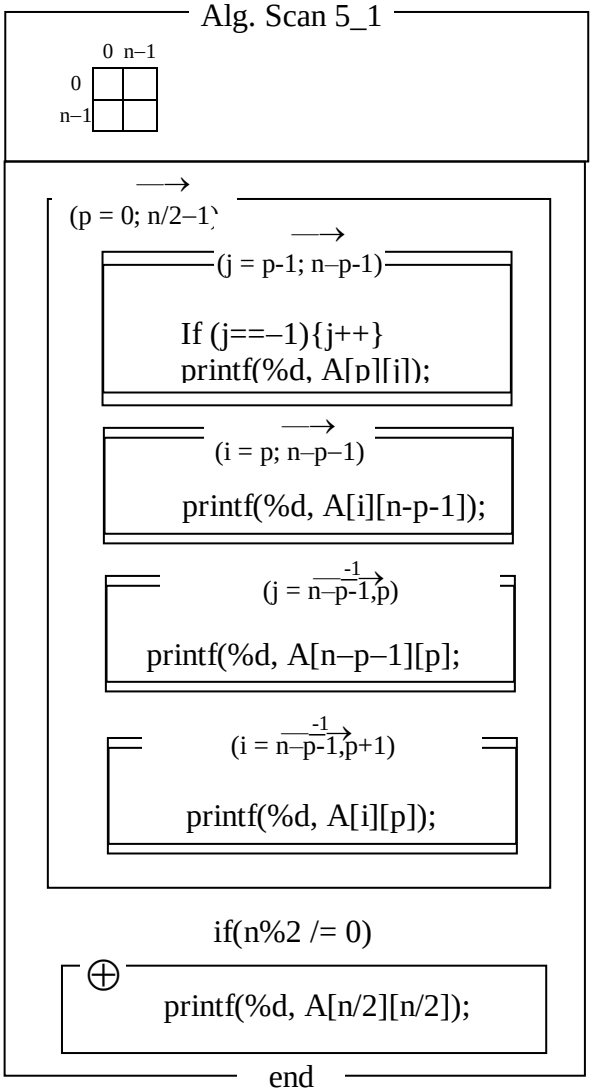
i = 0..n-1 n-1..0 0..n-1 n-1..0



## Обхід по спіралі




Змінна p — номер витка спіралі, починаючи з 0

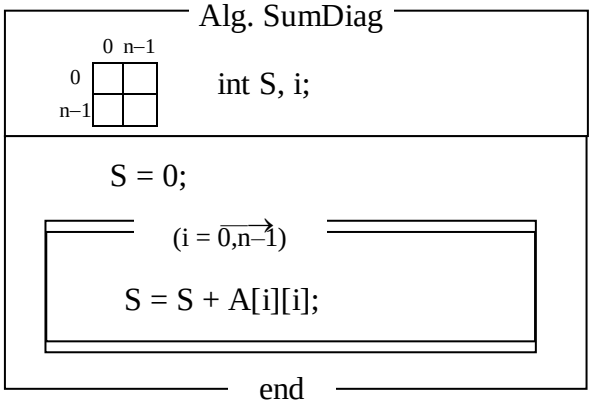


# Особливості роботи з квадратними матрицями

Діагональ матриці (головна)

| | *j*=0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| *i*= 0 | 0,0 | | | | | |
| 1 | | 1,1 | | | | |
| 2 | | | 2,2 | | | |
| 3 | | | | 3,3 | | |
| 4 | | | | | 4,4 | |
| *n*–1=5 | | | | | | 5,5 |

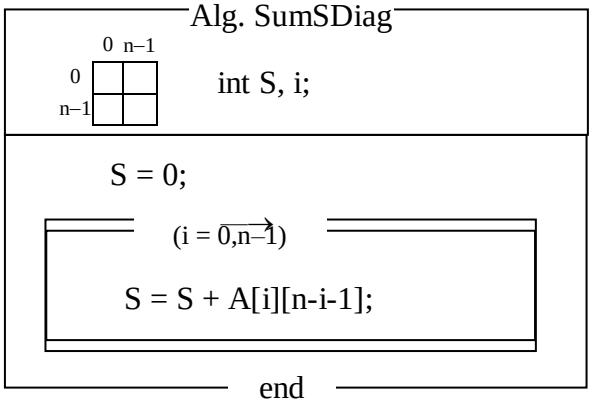
Задача: знайти суму елементів головної діагоналі



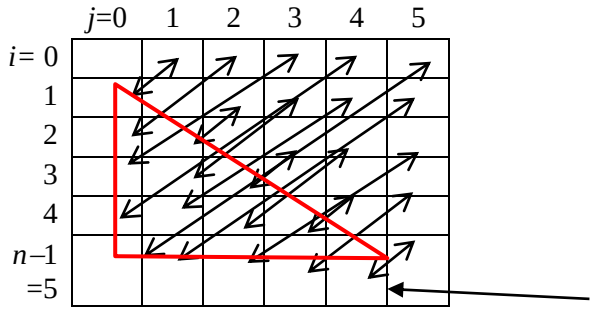

Побічна діагональ матриці

| | *j*=0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| *i*= 0 | | | | | | 0,5 |
| 1 | | | | | 1,4 | |
| 2 | | | | 2,3 | | |
| 3 | | | 3,2 | | | |
| 4 | | 4,1 | | | | |
| *n*–1=5 | 5,0 | | | | | |

Задача: знайти суму елементів побічної діагоналі

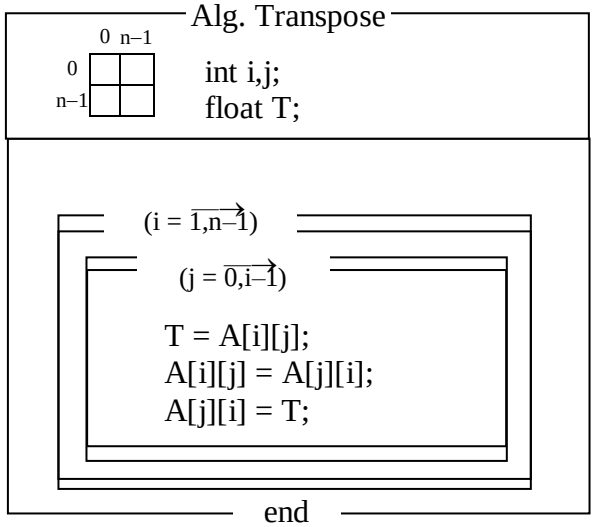



Транспонування матриці




A[i,j] <=> A[j,i];

Трикутник, який необхідно обійти;



# Лекція 12
# Алгоритми сортування

**Сортування** — упорядкування масиву за збільшенням або за зменшенням своїх значень. Значення — числове або порядку у множині (порядковий номер елементу).

Сортування широко використовується в базах даних для покращення пошуку в них. Тоді сортують структури: записи в базі з кількома полями. Вибирається поле, за яким відбувається сортування, наприклад, порядковий номер, числове значення параметру (дата народження), ім'я (за алфавітом).

Бази даних та їх сортування — одне з перших застосувань комп'ютерів, яке почало давати реальний прибуток від використання комп'ютерів у 50-х роках.

**Алгоритми сортування:**

— алгоритми, що використовують додаткову пам'ять,

— алгоритми, що НЕ використовують додаткову пам'ять (сортування на тому самому місці).

Перші алгоритми використовуються для сортування файлів та невеликих масивів.

Другі алгоритми – для сортування великих масивів та іноді – файлів прямого доступу.

**Алгоритми сортування на тому самому місці:**

1) Сортування вставкою (by insertion)
2) Сортування вибором (by selection)
3) Сортування обміном (by exchange)

Є прямі методи та покращені методи. Покращені методи використовують деякі оригінальні ідеї, які суттєво покращують сортування.

Прямі методи, як правило, не використовуються для вирішення задач, що є критичними за швидкодією.

Для невибагливих до швидкодії задач — вони використовуються часто.

У деяких окремих випадках —прямі методи також можуть сортувати краще за покращені методи.

**Оцінювання алгоритмів сортування**

Для оцінювання алгоритмів сортування використовують 2 показники:

1. Кількість операцій присвоювання або перестановок R (rearrange).
2. Кількість порівнянь C (compare).

При оцінці складності алгоритму використовують параметр — довжину масиву $n$. Наприклад, $C = O(n^2)$, $R = O(n \log n)$.

**Сортування методом прямої вставки**

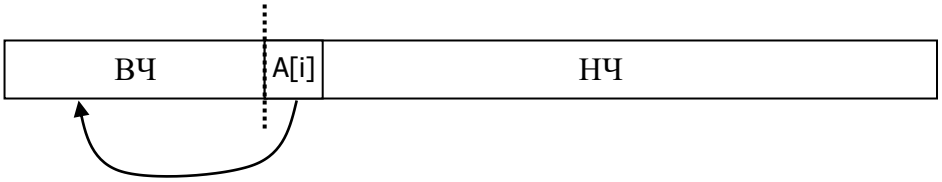
Принцип методу. В кожен момент часу масив вважається поділеним на дві частини:

1) вже відсортована частина (ВЧ);
2) ще невідсортована частина (НЧ).

**Алгоритм**

ВЧ = A[0].

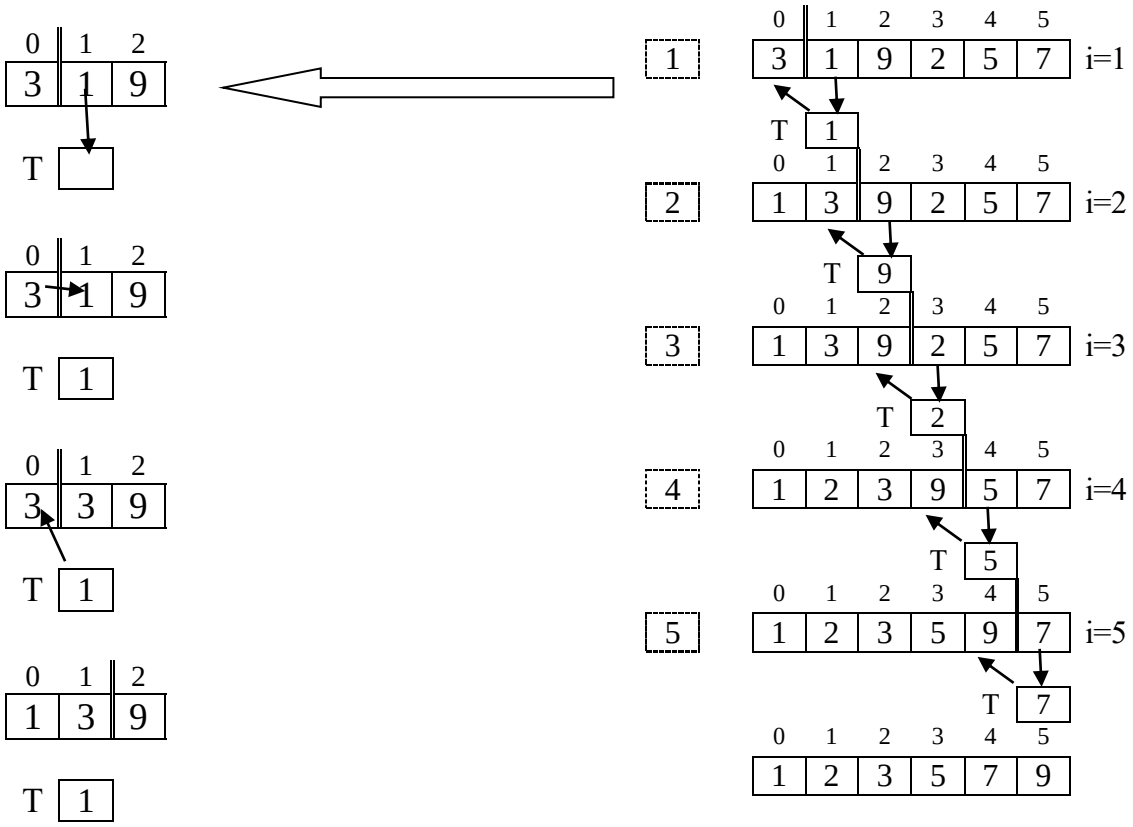
1. Береться черговий елемент A[i] з НЧ.
2. Шукаємо в ВЧ позицію, куди можна вставити елемент A[i] так, щоб відсортованість ВЧ не порушилась.
3. Звільняємо цю позицію за допомогою зсуву частини масису на 1 елемент.
4. Вставляємо A[i] у відсортовану частину на звільнене місце. Як результат, НЧ зменшилась на 1 елемент, а ВЧ — збільшилась на 1 елемент.
5. ПП 1–4 повторюють для усіх елементів з НЧ, поки довжина НЧ більше 0.




Для реалізації прямого методу вставки можна використати декілька модифікованих алгоритмів, які відрізняються різними алгоритмами пошуку місця вставки. Це наступні 4 алгоритми:

1. Алгоритм сортування прямою вставкою з лінійним пошуком від початку масиву (зліва).
2. Алгоритм сортування прямою вставкою з лінійним пошуком від елементу, що вставляється (справа).
3. Алгоритм сортування прямою вставкою з лінійним пошуком від елементу, що вставляється з використанням бар’єру (справа з бар’єром).
4. Алгоритм сортування прямою вставкою з двійковим пошуком.

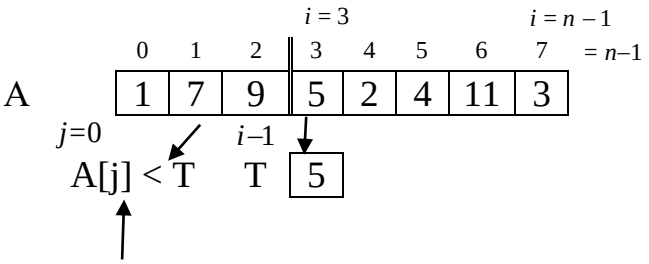
**Схема сортування прямою вставкою**




Кількість проходів — $n-1$

**Алгоритм 1**.

Сортування прямою вставкою з лінійним пошуком від початку масиву (зліва).

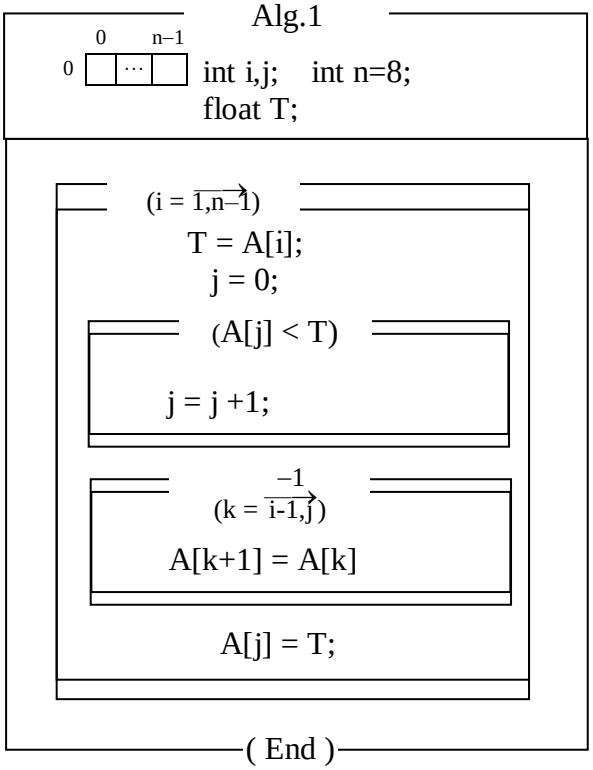



Умова продовження циклу

Пошук місця, куди вставити

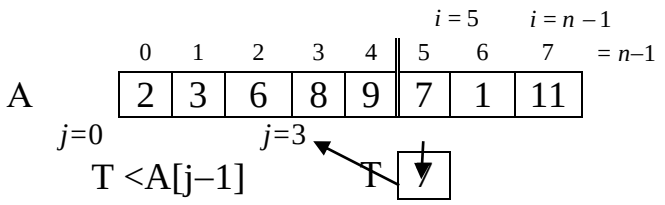
Зсув частини масива щоб вивільнити $j$ – е місце

Вставка відсортованого елементу



**Алгоритм 2**.

Сортування прямою вставкою з лінійним пошуком від елементу, що вставляється (справа).

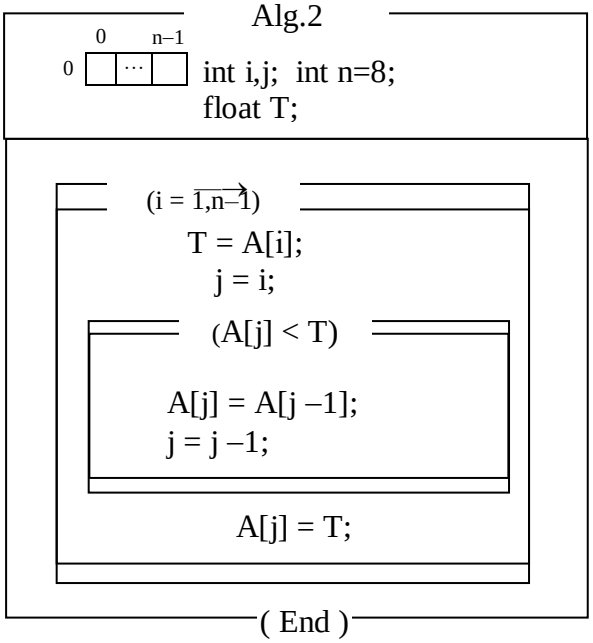



Умова продовження циклу

Тут цикли пошуку і зсуву суміщаються.

Місце *j*, куди вставити, вільне.

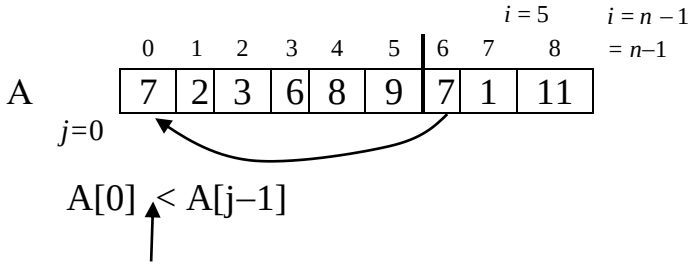
Вставка відсортованого елементу




**Алгоритм 3**.

Сортування прямою вставкою з лінійним пошуком від елементу, що вставляється з бар'єром (справа з бар'єром).

Сортується масив A[1]…A[n], комірка A[0] — вільна.

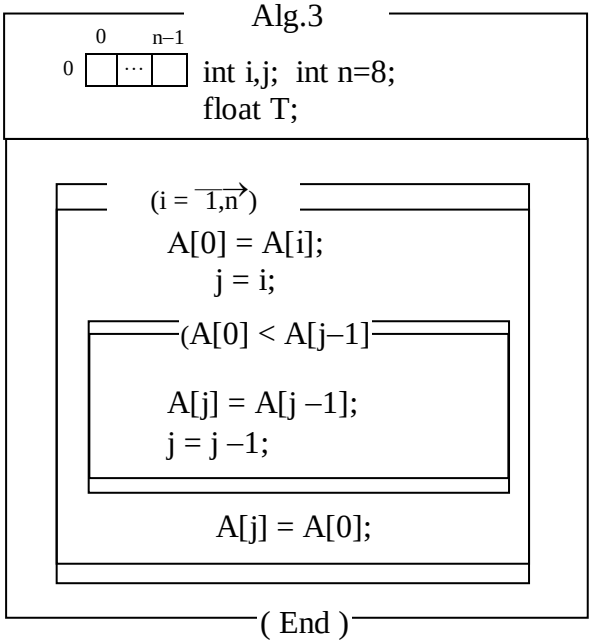



Умова продовження циклу

Тут цикли пошуку і зсуву суміщаються.

Місце *j*, куди вставити, вільне.

Вставка відсортованого елементу




У тих випадках, коли значеня для вставки меньше за A[1], комірка A[0] буде служити "бар'єром", щоб індекс j не вийшов за нижню границю масиву. Крім того, компонента A[0] замінює собою додаткову змінну T.

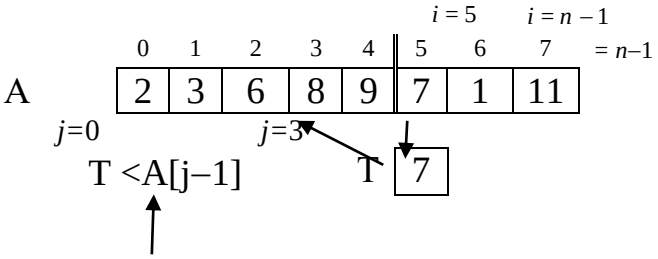
**Алгоритм 4**.

Сортування прямою вставкою з двійковим пошуком.

Для такого пошуку використовують алгоритм двійкового пошуку, який знаходить елемент , що стоїть справа від співпадаючого.

При цьому умова A[j] < T замінюється на умову A[j] < = T. Такий модифікований алгоритм буде зупинятися на індексі L =R, який вказує на елемент, розташований справа від співпадаючого або на більший з двох елементів, між якими міг би знаходитись шуканий елемент.

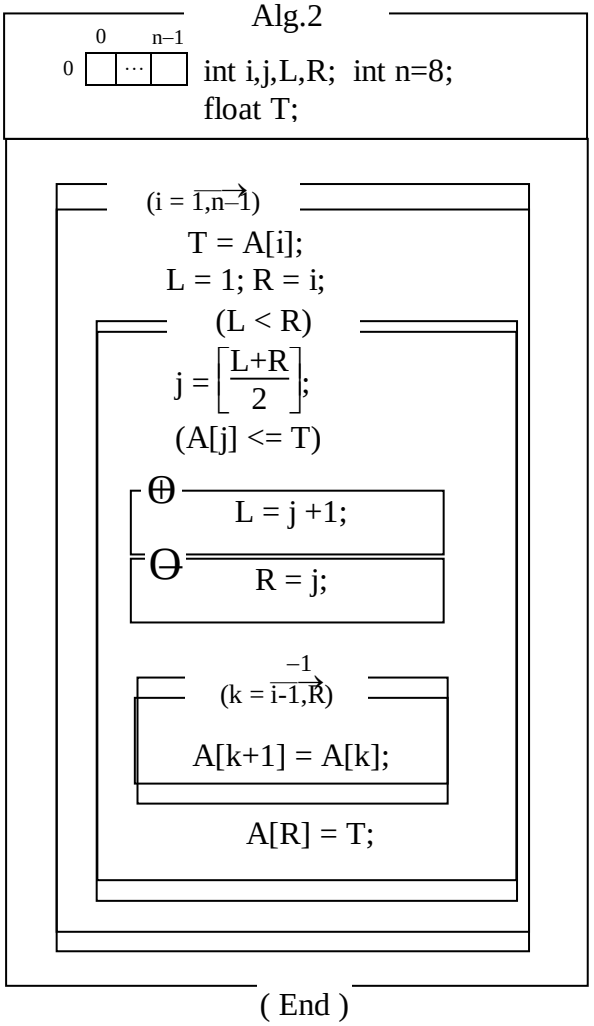
А якщо елемент є останнім в області пошуку, то вказівник буде вказувати безпосередньо на цей елемент, тобто, вказівник не вийде за межі масиву.




Умова продовження циклу

Тут цикли пошуку і зсуву суміщаються.

Індекс L = R показує місце вставки



**Аналіз алгоритмів прямої вставки**

Для всіх 4-х алгоритмів загальний час роботи має квадратичну характеристику.

Cmin=n-1; Rmin=3(n-1);

Cmax=$(n^2+n-4)/4$; Rmax=$(n^2+3n-4)/2$.

Алгоритм 4 має таку саму характеристику, що і інші, незважаючи на використання війкового пошуку.

Алгоритми 2 і 3 за поведінкою і характеристиками майже не відрізняються.

**Крайні ситуації**

Ситуація 1. Упорядкований масив

A1: $C = C_{max}$, $R = R_{min}$.

A2,A3: $C = C_{min}$, $R = R_{min}$.

A4: $C = (C_{max} + C_{min})/2$, $R = R_{min}$.

Ситуація 2. Обернено упорядкований масив

A1: $C = C_{min}$, $R = R_{max}$.

A2,A3: $C = C_{max}$ $R = R_{max}$.

A4: $C = (C_{max} + C_{min})/2$, $R = R_{max}$.

Відміна між кращим і гіршим випадками не така велика, як для A2,A3. Присвоювання практично на усіх комп'ютерах виконуються повільніше, ніж порівняння. Тому вага характеристики R більша за C.

# Лекція 13

## Алгоритми сортування (продовження)

**Алгоритм сортування прямим вибором**

В певному сенсі алгоритм протилежний сортування прямої вставки.

***При прямій вставці*** на кожному кроці розглядається тільки **один** черговий елемент вхідної послідовності і **всі** елементи готової послідовності для знаходження місця включення.

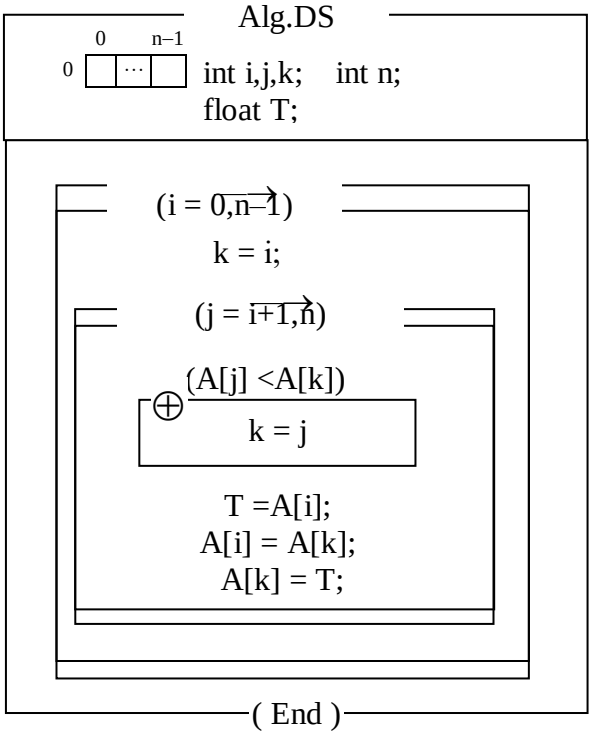
***При прямому виборі*** для пошуку одного елемента з найменшим ключем проглядаються **всі** елементи вхідної послідовності і знайдений **один** елемент поміщається як черговий елемент в кінець готової послідовності.

Метод сортування прямим вибором заснований на наступних правилах:

— Вибирається найменший елемент.

— Він міняється місцями з першим елементом a0.

— Потім ці операції повторюються з рештою $n-1$ елементами, $n-2$ елементами і так далі до тих пір, поки не залишиться один, найбільший елемент.




Число порівнянь C не залежить від порядку даних:

$$C=(n^2-n)/2.$$

Число перестановок мінімально при впорядкованих даних

$$R_{min} = 3(n-1)$$

І максимально, якщо порядок обернений

$$R_{max} = n^2/4 +3(n-1),$$

Середнє число пересилок

$$R_{ср} \approx n(\ln n + g),$$

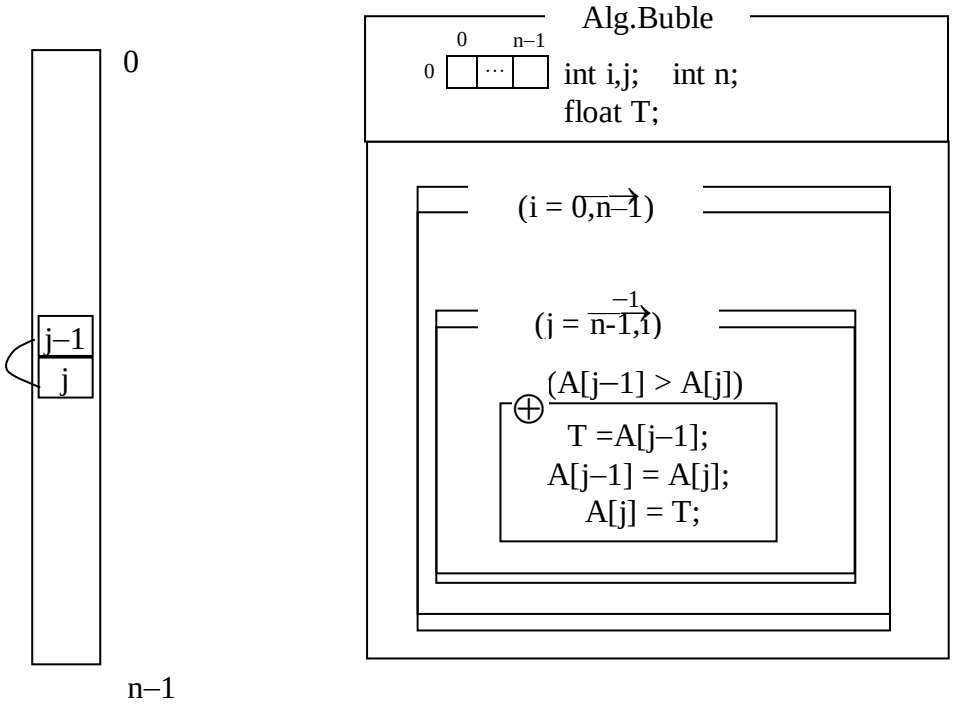
де g = 0,577216... — константа Ейлера.

Як правило, сортування прямим вибором краще алгоритму прямої вставки, однак, якщо дані на початку впорядковані або майже впорядковані, пряма вставка виявляється швидше.

**Алгоритм сортування прямим обміном**

Алгоритм ґрунтується на принципі порівняння і обміну пари сусідніх елементів до тих пір, поки не будуть відсортовані всі елементи. Як і в методі А1 відбуваються проходи по масиву, зсовуючи кожного разу найменший елемент послідовності, що залишилася, до початку масиву.

Якщо розглядати масиви як вертикальні, а не горизонтальні структури, то елементи можна інтерпретувати як бульбашки у склянці з водою, причому вага кожної з них відповідає величині його даного. В цьому випадку при кожному проході одна бульбашка наче піднімається до рівня, що відповідає її вазі. Такий метод відомий під назвою «бульбашкове сортування».




Число порівнянь в алгоритмі прямого обміну

$C = (n^2 - n) / 2,$

а мінімальне, середнє і максимальне число переміщень елементів відповідно:

$R_{min} = 0,$

$R_{ср} = 3 (n^2 - n) / 4,$

$R_{max} = 3 (n^2 - n) / 2.$

Отже, сортування прямим обміном являє собою щось середнє між сортуванням за допомогою вставок і за допомогою вибору;

Фактично у бульбашкового сортування немає нічого цінного, крім привабливого назви.

**Шейкер-сортування**

Шейкер-сортування є вдосконаленим методом бульбашкового сортування. Аналізуючи метод бульбашкового сортування, можна відзначити дві обставини:

— Якщо при русі по частині масиву перестановки не відбуваються, то ця частина масиву вже відсортована і, отже, її можна виключити з розгляду.

— При русі від кінця масиву до початку мінімальний елемент "спливає" на першу позицію, а максимальний елемент зсувається тільки на одну позицію вправо.

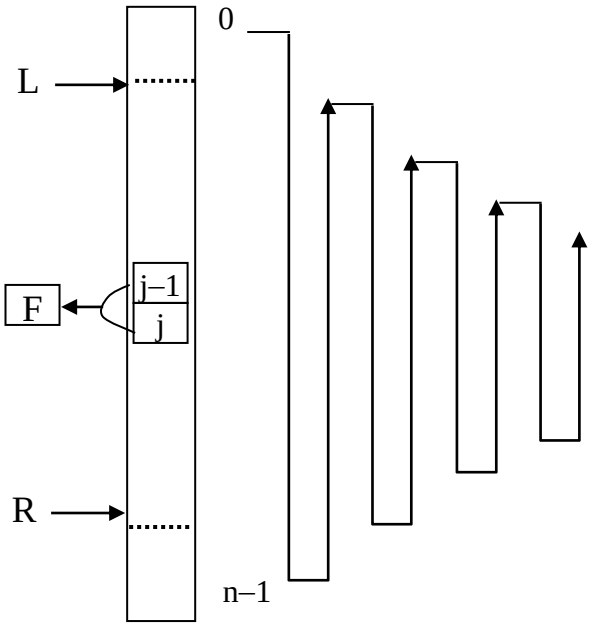
Ці дві ідеї призводять до модифікацій в методі бульбашкового сортування:

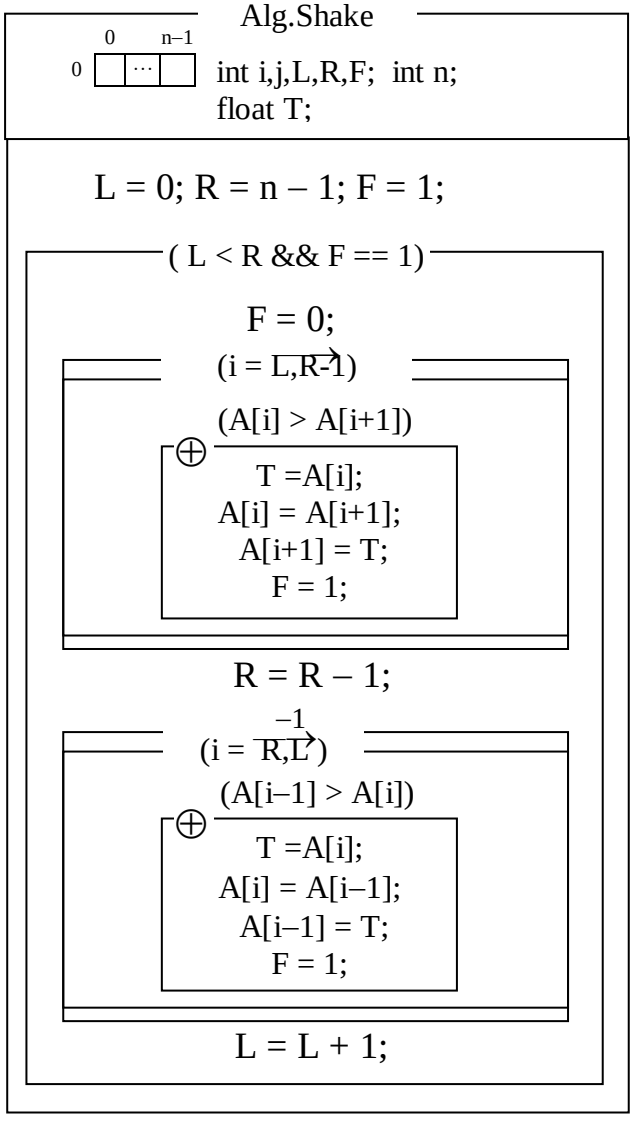
— Від останньої перестановки до кінця масиву знаходяться відсортовані елементи. З огляду на цей факт, перегляд здійснюється не до кінця масиву, а до конкретної позиції. Межі відсортованої частини масиву зсуваються на 1 позицію на кожній ітерації.

— Масив проглядається по черзі справа наліво і зліва направо.

— Перегляд масиву здійснюється до тих пір, поки всі елементи не встануть в порядку зростання (спадання).

L — ліва границя
R — права границя
F — прапорець переміщення, коли вже все відсортоване, а L < R та F = 0, то — кінець.



**Алгоритм Шела**

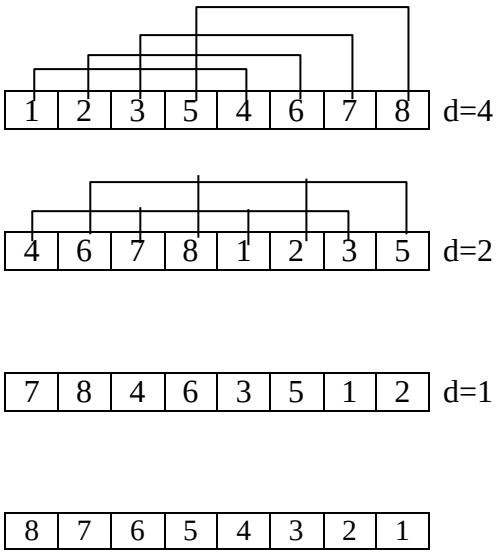
У 1959 р Д. Шелом було запропоновано удосконалення сортування за допомогою прямої вставки. Воно ґрунтується на тому, що масив з частково відсортованими елементами сортується краще.

Спочатку окремо групуються і сортуються елементи, віддалені один від одного на d = 4 позиції. Такий процес називається четвертним сортуванням.

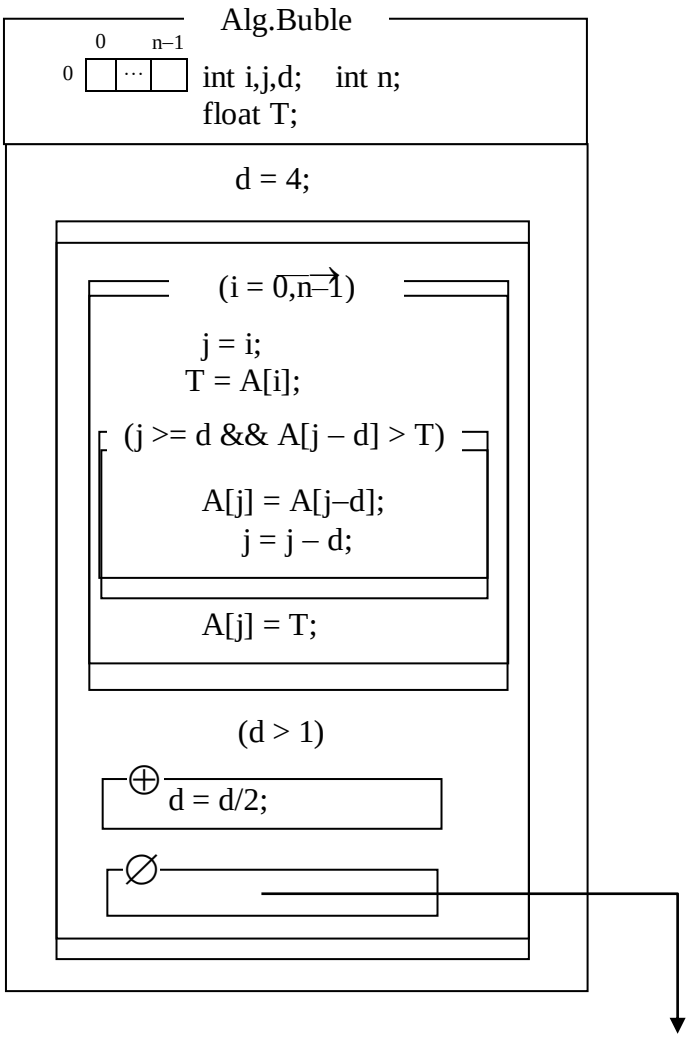
Після першого проходу елементи перегруповуються — тепер кожен елемент групи відстоїть від іншого на d = 2 позиції — і знову упорядковано (2−сортування).

На третьому проході виконується звичайне сортування.

На кожному етапі сортується мало елементів — при четвертному сортуванні — *n*/4, і елементи вже досить добре впорядковані. Тому потрібно порівняно небагато перестановок.




Такий метод в результаті дає упорядкований масив, і кожен прохід від попередніх тільки виграє (так як d-сортування об'єднує дві групи, вже відсортованих 2d-сортуванням).




В загальному випадку, для усіх **t** відстаней:

$$\mathbf{d_1, d_2, ..., d_t},$$

виконується умова

$$d_t=1;$$

$$d_{i+1} < d_i.$$

Кожне d–сортування програмується як сортування прямою вставкою.

В алгоритмі Шела не відомо, які відстані дають кращі результати. Дональд Кнут рекомендував таку послідовність $d_i$:

1, 3, 7, 15, 31, ...,

Деякі вчені рекомендують: 1, 4, 10, 23, 57, 132, 301, 701, 1750, далі множити на 2.25.

# Лекція 14

## Алгоритми сортування (закінчення)

### Алгоритм швидкого сортування

Швидке сортування є вдосконаленим метод сортування, заснований на алгоритмі обміну. Цей алгоритм винайшов Ч. Хоар і назвав його швидким сортуванням.

У масиві вибирається певний опорний елемент. Потім він поміщається в те місце масиву, де йому належить бути після упорядкування всіх елементів. В процесі відшукання відповідного місця опорного елементу переставляються елементи так, що зліва від нього знаходяться елементи, які менші за опроний, а справа — більші. (якщо масив сортується за зростанням).

Тим самим масив розбивається на дві частини:

— Невідсортовані елементи зліва від опорного елемента;

— Невідсортовані елементи праворуч опорного елемента.

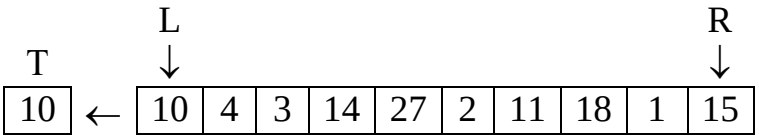
Щоб впорядкувати ці два менших подмассіва, алгоритм рекурсивно викликає сам себе.

Розглянемо сортування на прикладі масиву:

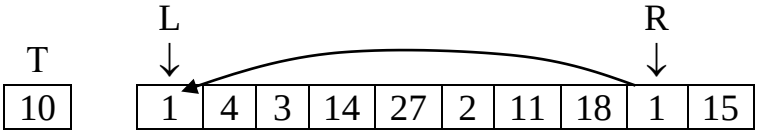
10, 4, 2, 14, 27, 2, 11, 18, 1, 15.

Для реалізації алгоритму переупорядкування використовуємо покажчик L на крайній лівий елемент масиву, який у процесі рухається вправр. Покажчик R поставимо на крайній правий елемент масиву, і він рухається вліво, поки елементи, на які він показує, залишаються більше за опорний.

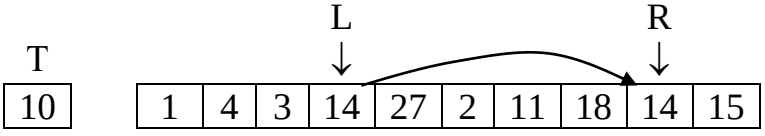
Нехай крайній лівий елемент – буде опорним. Встановимо покажчик L на наступний за ним елемент, а R — на останній. Алгоритм повинен визначити правильне положення елемента 10 і по ходу поміняти місцями неправильно розташовані елементи.



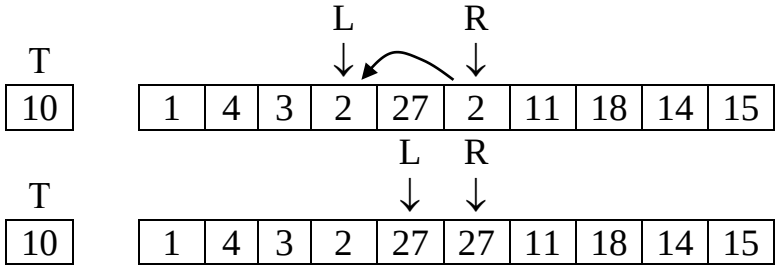

Рух покажчиків зупиняється, як тільки зустрічаються елементи, розташування яких щодо опорного елемента — неправильне.



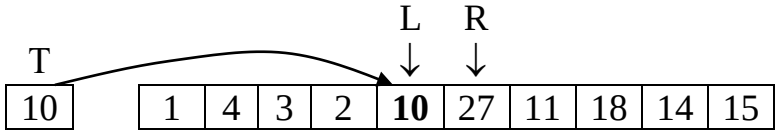

Покажчик L переміщується доки, поки не покаже елемент більше 10; R рухається, поки не покаже елемент менше за 10. Цей менший елемент, який стоїть на R, пересилається на місце L, а більший, що на L — пересилається на вивільне місце R.



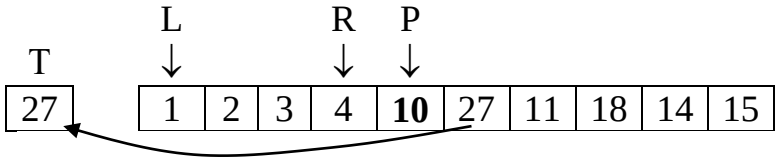
І рух покажчиків продовжується доки, поки R не виявиться зліва від L.

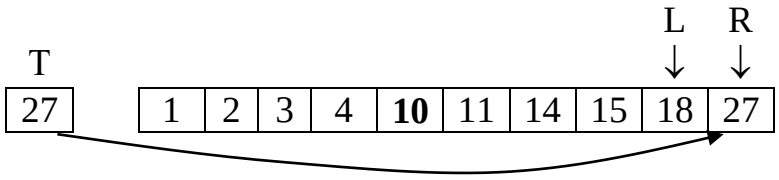



Тим самим визначається місце опорного елементу. Він міняється місцем з елементом, на який вказує R.



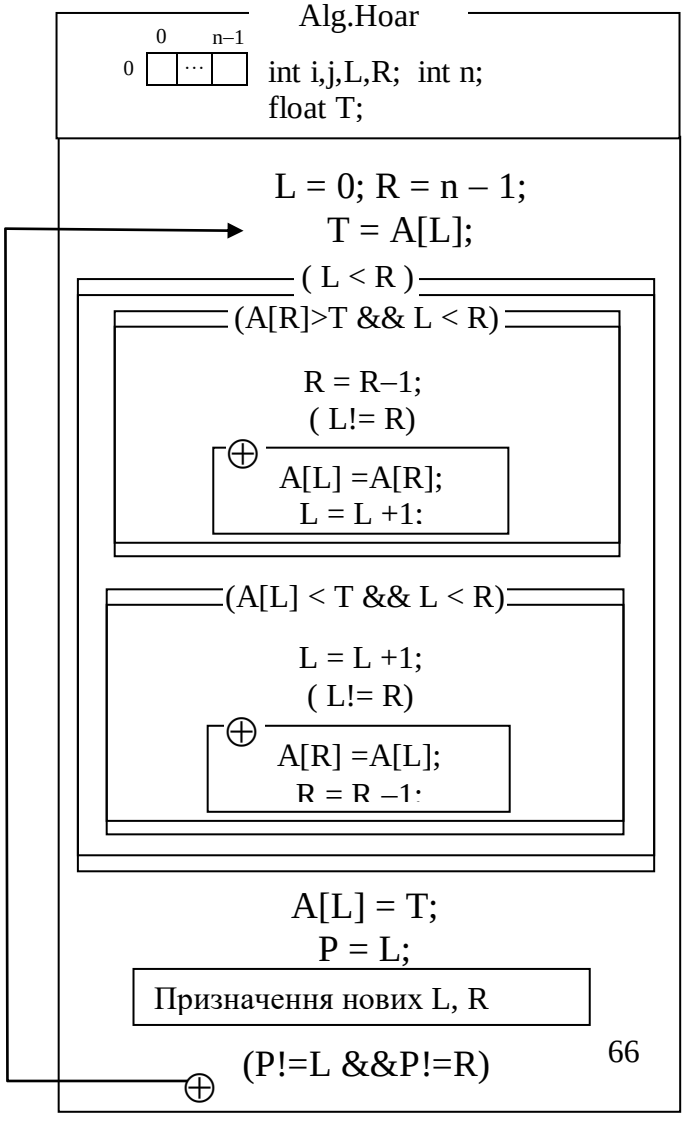

Після цього алгоритм викликається рекурсивно для лівої і правої частин відносно опорного елементу




В середньому, виконується
C = O(n log n) порівнянь та
R = O(n log n) пересилок

Для найгіршого випадку
C = O($n^2$);
R = O($n^2$);



**Сортування злиттям**

merge sort.

Часто використовується, коли неможливо мати довільний доступ до елемента масиву A[i]. Злиття означає об'єднання двох (або більше) послідовностей в одну впорядковану послідовність за допомогою циклічного вибору елементів, які доступні в даний момент.

Процедура злиття передбачає об'єднання двох попередньо упорядкованих підпослідовностей розмірами n/2 в єдину послідовність розмірами n. Початкові елементи попередньо упорядкованих послідовностей порівнюються між собою, і з них вибирається найменший.

Процедура злиття повторюється рекурсивно для підмасивів довжиною 2, 4,...n/2 або навпаки, n/2, ..., 4, 2.

Наприклад:

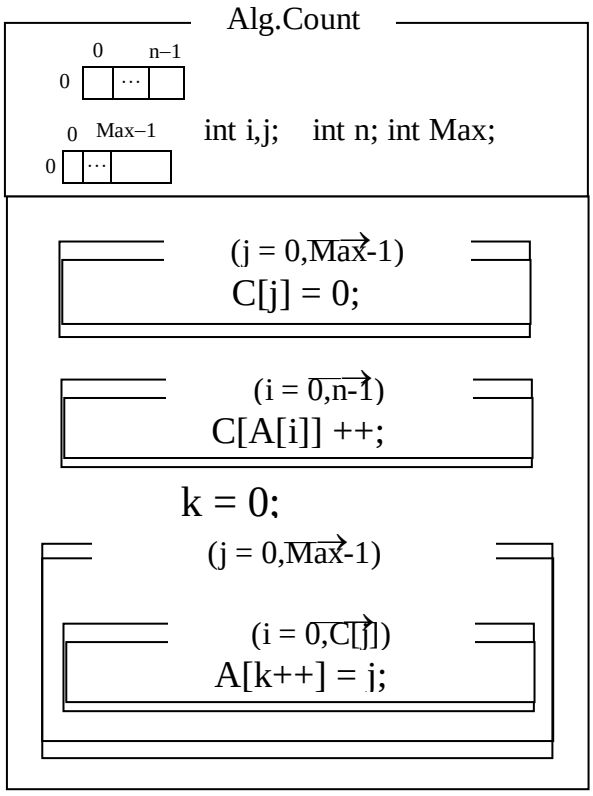
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 42 | 5 | 30 | 36 | 25 | 10 | 37 | 49 |
| 1 етап | 5 | 42 | 30 | 36 | 10 | 25 | 37 | 49 |
| 2 етап | 5 | 30 | 36 | 42 | 10 | 25 | 37 | 49 |
| 3 етап | 5 | 10 | 25 | 30 | 36 | 37 | 42 | 49 |

**Сортування з підрахунком**

Використовується для великих масивів цілих даних.

Нехай дані приймають значення від 0 до 7. Тоді заводиться масив C[8]. Вхідний масив сканується і C[j] = C[j] +1, якщо A[i] = j.

Після цього генерується відсортований масив. Починаючи з C[0], у масив A послідовно заноситься число j стільки разів, скільки показує число у комірці C[j].




Різновидом є **порозрядне сортування**.

Наприклад, спочатку сортують за старшим розрядом або групі розрядів, потім за середнім і т.д.

Наприклад, це сортування за алфавітом.

Для сортування – структура масиву записів, у якому теги – значення розрядів.

Отже, сортування з підрахунком — найшвидше сортування.

Але воно неефективне для масивів невеликого об’єму,

масивів з даними, які мають великий діапазон.

Також цей алгоритм вимагає великого додаткового обсягу пам’яті.

# Лекція 15
# Рядки

**Рядок** — це різновид одномірного масиву, елементи якого є символами. Він має певні особливості реалізації та поводження на відміну від звичайного масиву.

У різних мовах рядки представляються по-різному.

1) Турбо-рядок:

| 11 | Т | у | р | б | о | - | р | я | д | о | к |
|---|---|---|---|---|---|---|---|---|---|---|---|

У першому байті – довжина рядка. Наприклад, мова Pascal.

2) Рядок з завершальним нулем

| Т | у | р | б | о | - | р | я | д | о | к | Null |
|---|---|---|---|---|---|---|---|---|---|---|---|

У останньому байті – нуль. Наприклад, мова Сі.

У мові Сі визначення

char astr [10];

вказує компілятору на резервування місця для максимум 10 символів. Символьна адреса astr містить адресу комірки пам'яті, в якій вміщено значення першого з десяти об'єктів типу char.

Процедури, які заносять рядки в масив astr, копіюють його по одному символу в область пам'яті, на яку вказує astr, до тих пір, поки не буде скопійований нульовий символ, який завершає рядок.

Коли виконується функція типу

printf ( "%s", astr);

їй передається значення astr, тобто адреса першого символу, на який вказує astr. Якщо перший символ — нульовий, то робота функції printf () закінчується, а якщо немає, то вона виводить його на екран, додає до адресою одиницю і знову починає перевірку на нульовий символ.

Така обробка дозволяє зняти обмеження на довжину рядка, але в межах доступної пам'яті.

Рядок можна ініціалізувати як:

```
char array[6] = "Рядок";
char s[ ] = {'Р', 'я', 'д', 'о', 'к', '\0'};
```

(При визначенні масиву з одночасною ініціалізацією межі зміни індексу можна не вказувати).

Другий спосіб завдання рядка — це покажчик на символ. Визначення

char * b;

задає змінну b, яка може містити адресу деякого об'єкту. Однак в даному випадку компілятор резервує місце для зберігання символів і не ініціалізує змінну b конкретним значенням. Коли компілятор зустрічає оператор

b = "Рядок" ;

він виробляє такі дії. По-перше, він створює в будь-якому місці об'єктного модуля рядок "Рядок", за яким слідує нульовий символ ( '\0').

По-друге, він призначає початкову адресу цього рядка (адреса символу 'Р') змінній b.

Функція

printf ( "%s", b);

виводить символи до тих пір, поки не зустрінеться заключний нуль.

Масив покажчиків можна форматувати, тобто призначати його елементам конкретні адреси деяких заданих рядків при визначенні.

У бібліотеці, що приєднується за stdio.h, є кілька корисних функцій роботи з рядками. Для введення і виведення рядків символів крім scanf () і printf () можуть використовуватися функції gets () і puts ().

Якщо string - масив символів, то ввести рядок з клавіатури можна так:

gets(string);

і закінчити натисканням клавіши <Enter>. Вивести рядок на екран можна так:

puts(string);

У бібліотеці, що приєднується за string.h є кілька корисних функцій роботи з рядками. Найчастіше використовуються функції strcpy( ), strcat( ), strlen( ) и strcmp( ).

strcpy(string1, string2);

– служить для копіювання рядка string2 у рядок string1. Масив string1 має бути досить великим, щоб у нього умістився рядок string2. Так як компілятор не слідкує за такою ситуацією, то недостача місця приведе до втрати даних.

strcat(string1, string2);

— приєднує рядок string2 до рядка string1 і кладе його у масив, де був рядок string1, а рядок string2 не змінюється. Нульовий байт, що закінчував перший рядок, заміняється першим байтом другого рядка.

Функція strlen( ) повертає довжину рядка, а завершальний нульовий байт не приймається до уваги:

int a = strlen(string);

Функція strcmp( ) порівнює два рядки і вертає 0, якщо вони однакові.

# Лекція 16
# **Записи**

У Pascal, Ada — record (запис),
У C, C++, — struct (структура).
Запис на відміну від масиву, файлу є **неоднорідною** структурою даних.
Він може включати в себе довільну к-ть структур даних інших типів.
Складові запису наз-ся **полями**.
У запису є головний ідентифікатор,
У полів є власні ідентифікатори.
Щоб дістатися поля, вказують ім'я запису і через крапку — ім'я поля.
Такий ідентифікатор називають складеним (qualified identifier).

Наприклад, об'ява структури:

```
sruct date { int day;
             int month;
             int year;
           };
```

Після неї можна призначити змінним тип цієї структури:

```
struct date {...} a, b, c;
```

При цьому змінним виділяється пам'ять.
Пам'ять виділяється і призначається ідентифікатор потім:

```
struct date days;
```

Запис можна ініціалізувати, поміщаючи у кінці список елементів:

```
struct date days= {21, 11,2017};
```

Можна вкладати записи один в одний:

```
struct person{ char name1[20], name2[20];
               struct date bday;
               int age;
             };
```

Приклад використання:

```
struct person man[100];
```

Тут визначено масив man, що складається з 100 структур типу man.

Щоб звернутися до окремого елементу структури, необхідно вказати його ім'я, поставити крапку і відразу ж за нею записати ім'я потрібного елемента, наприклад:

```
man[j].age = 19;
man[j].bday.day = 24;
man[j].bday.month = 2;
man[j].bday.year = 1997;
```

При роботі зі структурами необхідно пам'ятати, що тип елемента визначається відповідним рядком опису в фігурних дужках. Наприклад, масив man_ має тип man, year є цілим числом і т.п. Оскільки кожен елемент структури відноситься до певного типу, його ім'я може з'явитися скрізь, де дозволено використання значень цього типу. Допускаються конструкції виду

```
man_ [i] = man_ [j];
```

де man_ [i] і man_ [j] — об'єкти, відповідні єдиному опису структури. Іншими словами, дозволяється присвоювати одну структуру інший за їхніми іменами.

Унарна операція & дозволяє взяти адресу структури. Припустимо, що визначена змінна day:

```
struct date {int d, m, y;} day;
```

Тут day - це структура типу date, що включає три елементи: d, m, y. Інше визначення

```
struct date * bdate;
```

встановлює той факт, що db - це покажчик на структуру типу date.

Запишемо вираз:

```
db = & day;
```

В цьому випадку для вибору елементів d, m, y структури необхідно використовувати конструкції:

```
(* db) .d; (* db) .m; (* db) .y;
```

Дійсно, db — це адреса структури, * db - сама структура. Круглі дужки тут необхідні, так як точка має більш високий пріоритет, ніж зірочка. Для аналогічних цілей в мові Сі передбачена спеціальна операція ->. Ця операція вибирає елемент структури і дозволяє уявити розглянуті вище конструкції в більш простому вигляді:

```
db -> d; db -> m; db -> y;
```

# Множини

**Перелічний тип даних**

Перелічний тип даних призначений для опису об'єктів з деякої заданої множини. Перелічний тип задає тип, який є підмножиною цілого типу.

Він задається ключовим словом enum. Рассморім приклад:

```
enum seasons {spring, summer, autumn, winter};
```

Тут введено новий тип даних seasons. Приклад:

```
enum boolean {true = 1, false = 0};
```

Тут 0 та 1 – це константні вирази, які мають бути цілими числами. За замовчуванням, якщо не задано константний вираз, першому елементу присвоюється значення 0, наступного елементу — значення 1 і т.д.

Наступний елемент списку отримує значення, яке дорівнює <константний вираз> + 1, якщо тільки його значення не задається явно іншим константним виразом.

У списку переліку можуть міститися елементи, яким зіставлені однакові значення, проте кожен ідентифікатор в списку повинен бути унікальним. Крім того, ідентифікатор елемента списку переліку повинен

бути відмінним від ідентифікаторів елементів всіх інших списків переліків, а також від інших ідентифікаторів.

Тепер можна визначити змінні цього типу:

enum seasons a, b, c;

Кожна з них (a, b, c) може приймати одне з чотирьох значень: spring, summer, autumn і winter. Ці змінні можна було визначити відразу при описі типу:

enum seasons (spring, summer, autumn, winter) a, b, c;

Розглянемо ще один приклад:

enum days {mon, tues, wed, thur, fri, sat, sun} my_week;

Імена, занесені в days (також як і в seasons в попередньому прикладі), являють собою літералі цілого типу. Перша з них (mon) автоматично встановлюється в нуль, і кожна наступна має значення на одиницю більше, ніж попередня (tues = 1, wed = 2 і т.д.).

Можна привласнити константам певні значення цілого типу (іменах, які не мають їх, будуть, як і раніше, призначені значення попередніх констант, збільшені на одиницю). наприклад:

enum days (man = 5, tues = 8, wed = 10, thur, fri, sat, sun} my_week;

Після цього mon = 5, tues = 8, wed = 10, thur = 11, fri = 12, sat = 13, sun = 14.

Тип enum можна використовувати для завдання констант true = 1 і false = 0, наприклад:

enum t_f (false, true) a, b;

Використовуйте перерахування тільки там, де це має сенс. А де це має сенс? Як правило скрізь, де використовуються константи, наприклад, в операторах switch.

**Об'єднання**

Об'єднання дозволяє в різні моменти часу зберігати в одному об'єкті значення різних типів. У процесі оголошення об'єднання з ним асоціюється набір типів значень, які можуть зберігатися в даному об'єднанні. У кожен момент часу об'єднання може зберігати значення лише одного типу з набору. Контроль над тим, якого типу значення зберігається в даний момент в об'єднанні, покладається на програміста.

union [<тег>] {<список оголошень елементів>} <опис> [, <опис> ...];
union <тег> <опис> [, <описатель> ...];

Тег призначений для розрізнення декількох об'єднань, оголошених в одній програмі.

Пам'ять, яка виділяється для змінної типу об'єднання, визначається розміром найдовшого елемента об'єднання. Всі елементи об'єднання розміщуються в одній і тій самій області пам'яті з однією адресою. Значення поточного елемента об'єднання втрачається, коли присвоюється значення іншого елементу об'єднання.

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

```
void main()
 {  union
    { float    f;
      long int i;
     } u;
   printf("Input float number: "); scanf("%f", &u.f);
   printf("Internal: %08x\n\n", u.i);

       }
```