Основи розробки програм

Viacheslav Davydov (author)

Зміст

1 Вступ до програмування

1.1 Визначення

Програмування – процес створення комп'ютерних програм. Ключовими безпосередніми завданнями програмування є створення та використання алгоритмів та структур даних. У більш широкому сенсі під програмуванням розуміють увесь спектр діяльності, пов'язаний зі створенням і підтримкою в робочому стані програм – програмного забезпечення. Ця інженерно-технічна дисципліна називається «програмна інженерія». Сюди входять:

- аналіз і постановка задачі,
- проектування програми,
- розробка алгоритмів та розробка структур даних,
- написання текстів програм,
- налагодження і тестування програми (випробування програми),
- документування,
- налаштування (конфігурація),
- доробка та супровід розробленого програмного продукту.

Програмування ґрунтується на використанні мов програмування, на яких записуються інструкції для комп'ютера. Сучасні додатки містять безліч таких інструкцій, пов'язаних між собою.

Мова програмування – формальна знакова система, яка призначена для запису комп'ютерних програм. Мова програмування визначає набір лексичних, синтаксичних і семантичних правил, що визначають зовнішній вигляд програми та дії, які виконає виконавець (зазвичай – ЕОМ) під її керуванням.

З часу створення перших програмованих машин людство придумало більше ніж вісім тисяч мов програмування (включаючи нестандартні, візуальні та езотеричні мови). Щороку їх кількість збільшується. Деякими мовами вміє користуватися тільки невелика кількість людей – їх власних розробників, інші стають відомими мільйонам.

Як правило, мова програмування існує у вигляді:

- стандарту мови – набору специфікацій, що визначають його синтаксис і семантику; стандарт мови може історично розвиватися (наприклад, ANSI C, C11);
- реалізацій (втілення) стандарту – власне програмних засобів, що забезпечують роботу відповідно до того чи іншого варіанту стандарту мови; такі програмні засоби розрізняються по виробнику, марці та варіанту (версії), часу випуску, повноті втілення стандарту, додаткових можливостей; можуть мати певні помилки або особливості втілення, які впливають на практику використання мови або навіть на його стандарт (наприклад, Borland C, MSVC, GCC).

1.2 Класифікація мов програмування

Наявні мови програмування (МП) прийнято класифікувати за кількома такими параметрами:

1. по ступеню орієнтації на специфічні можливості ЕОМ мови програмування діляться на:
    - машинно-залежні;
    - машинно-незалежні;
2. по ступеню деталізації алгоритму одержання результату:
    - мови низького рівня (асемблер);
    - мови високого рівня (C, Паскаль, C#);
    - мови надвисокого рівня. (Icon, APL, Haskell). На відміну від МП високого рівня, де описується принцип «як потрібно зробити», у надвисокорівневих МП описується лише принцип «що потрібно зробити». До цієї групи відносять мови логічного та функціонального програмування;
3. по ступеню орієнтації на розв'язок певного класу завдань:
    - проблемно-орієнтовані (TeX, SQL, HTML, Mapple);
    - універсальні (C, Паскаль, Java);
4. по можливості доповнення новими типами даних і операціями:
    - розширювані;
    - не розширювані (Fortran);
5. по можливості керування реальними об'єктами та процесами:
    - мови систем реального часу (C, асемблер);
    - мови систем умовного часу;
6. по способу одержання результату:
    - процедурні;
    - непроцедурні;
7. по типу розв'язуваних завдань:
    - мови системного програмування (C, асемблер);
    - мови прикладного програмування (Java, C#, Ruby);
8. непроцедурні мови по типу вбудованої процедури пошуку розв'язків діляться на:
    - реляційні (SQL);
    - функціональні (List, Haskell);
    - логічні (Prolog).

Процедурні (імперативні) МП. Характеризуються тим, що за допомогою механізмів самої мови послідовно виконувані оператори можна зібрати в підпрограми, тобто більші цілісні одиниці коду. Процедурна МП надає можливість програмістові визначати кожний крок у процесі розв'язку завдання. Особливість таких МП полягає в тому, що завдання розбиваються на кроки і вирішуються крок за кроком. Використовуючи процедурну мову, програміст визначає мовні конструкції для виконання послідовності алгоритмічних кроків.

Машинно-залежні (низького рівня) МП. Приклади таких мов: машинні (бінарний код), асемблер, автокоди. Використовуються в системнім програмуванні. Програма машинно-залежною мовою програмування може виконуватися тільки на ЕОМ даного типу.

Машинно-незалежні (високого рівня) МП. Програма машинно-незалежною МП після перетворення на машинну мову стає машинно-залежною. Ця ознака МП визначає мобільність одержуваних програм (можливість переносу на ЕОМ іншого типу).

Декларативні МП. У декларативнім програмуванні задається специфікація розв'язку завдання, тобто дається опис того, що являє собою проблема і який очікується результат. Програми, створені за допомогою декларативної мови, не містять змінних та операторів присвоювання. До декларативних мов можна віднести SQL і HTML. До підвидів декларативного програмування відносять функціональне і логічне програмування.

Функціональні мови програмування – мови програмування на основі лямда вирахувань, у яких процес обчислення трактується як обчислення значення функцій у математичнім розумінні. На відміну від імперативного стилю, що описує кроки, які ведуть до досягнення мети, функціональний стиль описує математичні відносини між даними і метою. Функціональні мови є мовами штучного інтелекту. Програма, написана функціональною мовою, складається з послідовності функцій і виразів, які необхідно обчислити. Основною структурою даних є зв'язний список. Функціональне програмування принципово відрізняється від процедурного. Основними функціональними мовами є Lisp, Miranda, Haskel.

Опис функціональних мов програмування є найскладнішим. Тому, наведемо приклад порівняння рішення однієї і тій же задачі випікання пирога з використанням імперативної та функціональної мови програмування.

1. Алгоритм дій випікання *імперативного* пірога:
    - Розігрійте духовку до 175° С. Змастіть маслом і посипте борошном деко. У маленькій мисці змішайте борошно, харчову соду і сіль.
    - У великій мисці збивайте масло, цукор-пісок і коричневий цукор до тих пір, поки маса не стане легкою і повітряної. Вбийте яйця, одне за раз. Додайте банани і розітріть до однорідної консистенції. По черзі додавайте в отриману кремову масу основу для тесту з п.1 і кефір. Додайте подрібнені волоські горіхи. Викладіть тісто в підготовлений лист.
    - Запікайте в розігрітій духовці 30 хвилин. Вийміть деко з духовки, поставте на рушник, щоб пиріг охолонув.
2. Алгоритм дій випікання *функціонального* пірога:
    - Пиріг - це гарячий пиріг, остиглий на рушнику.
    - Гарячий пиріг - це підготовлений пиріг, що випікався в розігрітій духовці 30 хвилин.
    - Розігріта духовка - це духовка, розігріта до 175° С.

- Підготовлений пиріг - це тісто, викладене в підготовлений лист.
- Тісто - це кремова маса, в яку додали подрібнені волоські горіхи.
- Кремова маса - це масло, цукор-пісок і коричневий цукор, збиті в великій мисці до тих пір, поки вони не стали легкими і повітряними.
- і т.д.

Логічні МП. Це такі мови, які орієнтовані на розв'язок завдань без опису алгоритмів, мови штучного інтелекту. Представником логічного програмування є Prolog, яким написана більшість експертних систем.

Об'єктно-орієнтовані мови стали подальшим рівнем розвитку процедурних мов, основною концепцією яких є сукупність програмних об'єктів. Написання програми мовою представляється у вигляді послідовності створення екземплярів об'єктів і використання їх методів. До них відносяться з перших мов Simula і Smalltalk, далі C++, Java.

1.3 Чому для викладання обрано мову С

Дискусії про поточне положення C/C++ у світі програмування, як правило, ділять учасників на два фронти: одні пророкують цій мові швидку загибель; інші ж навпаки стверджують, що на C/C++ писали, пишуть і будуть писати. Істина перебуває десь посередині. Але це означало б, що C/C++ перебуває в якому-то «підвішеному» стані, у якім вона була, наприклад, між виходами стандартів C++03 і C++11. Насправді все не так. А як? Спробуємо розібратися. Де хто вважає, що мова C/C++ на грані вимирання. Але:

1. Програмне забезпечення, яке написане з використанням C/C++, використовується і вимагає підтримки;
2. Нове ПО, наприклад, програмування ігрових движків також щосили використовує C++. А стандарти C++11, C++14, C++17 тільки підтверджують, що мова постійно розвивається і вмирати не збирається. Кожний новий стандарт несе в собі безліч доповнень, розширень і «синтаксичного цукру». Писати програми з використанням мови нових стандартів стає простіше. До того ж у стандартну бібліотеку додається безліч засобів, що прискорюють процес написання програм.
3. З використанням C/C++ написано безліч програмного забезпечення. І, звичайно, його необхідно підтримувати. До таких продуктів можна віднести:
    - ядра більшої частини сучасних операційних систем: Unix, Linux, Mac OS, Android, Windows Phone, Windows;
    - СУБД такі як MongoDB, Oracle, MySQL, MSSQL, PostgreSql;
    - ПО по роботі з графікою і відео, де потрібна висока продуктивність, наприклад, продукти компанії Adobe, Autodesk.

Навряд чи хтось стане переписувати існуючий працюючий C++ код на C# або Java, тільки тому, що так буде зручніше надалі, або ще чомусь. Таким чином виходить, що знання C++ необхідні. А з урахуванням того, що C++

став набагато зручнішим в роботі з виходом нових стандартів, ідея писати модулі/доповнення на сучасному C++ має місце бути. Надалі такий код буде куди простіше підтримувати. У той же час C++ – все та ж продуктивна мова програмування з великим набором можливостей, а доповнена стандартна бібліотека дозволить використовувати вже готові розв'язки без необхідності «винаходу велосипедів».

1.4 Чому мова C/C++ не популярна серед початківців

1. Високий поріг входження. Напевно, немає такого програміста, який би не чув про складність C++. Безумовно, мова велика, нюансів – багато. А що є натомість? Продуктивність + глибинний контроль процесів, які відбуваються (особливо якщо трохи спуститися з C++ до чистого C). Але тут знову слід згадати про стандарт C++11, який більш привітний до новачків, пропонуючи зручний синтаксис, різного роду контейнери, алгоритми та інші речі, призначені для того, щоб полегшити написання коду. Саме тому для вивчення в якості базової була обрана саме ця мова. Знання C/C++ допоможе освоїти інші суміжні мови з меншими витратами.
2. Слабке просування. На жаль мова C/C++ не має підтримки таких гігантів IT-індустрії, як Microsoft і Oracle, які проводять змагання і збори (такі як Хакатони) з упором на мови C# або Java.
3. Спеціалізація. Незважаючи на те, що C/C++ – мова, яка, як то кажуть, дає розробнику усі карти в руки, область її застосування не всеосяжна і займає певну нішу. Конкуренти є і це теж має своє значення. Узяти, наприклад, мобільну розробку. Основні платформи зайняті тими або іншими мовами: Windows Phone – C#, Android – Java, iOS – Objective-C/Swift. І це зовсім не означає, що під ці платформи немає можливості писати на C++, питання лише в тому, наскільки це буде зручно і чи буде ефективно. Для веб, розробка під який з кожним роком стає все популярнішою, C++ теж не дуже підходить.

З врахуванням того, що мобільна та веб-розробки сталі дуже популярні і тільки набирають оберти, зрозумілий вибір початківців не на користь C/C++.

1.5 Наскільки програмістові потрібно знання математики?

Багатьох студентів і просто початківців програмістів цікавить питання – «А чи потрібно вчити математику, щоб осягнути дзен програмування?». Однозначної відповіді на дане питання немає. Усе залежить від завдань, які будуть вирішуватися в майбутньому. Нижче наведені ключові моменти, які допоможуть одержати (або підштовхнуть) до відповіді на дане питання.

Математика забезпечує підґрунтя для програмування тому, що:

- математика закладає основи аналізу та побудови алгоритмічних моделей, а програмування – це автоматизація математичних дій;

- одне з основних умінь для математика – оперувати абстрактними поняттями, це уміння допомагає і програмістам у їхній роботі;
- математика – це базис, на якім будується ланцюг алгоритмів, основа будь-якої програми, яку програміст описує. Гарний програміст розуміє, що робить програма, розбирається в логіці і суті описуваних процесів. Тільки знання математики дозволить написати оптимальну програму;
- математика корисна для розвитку технічного і структурного мислення, потрібного для системного аналізу;
- знання математики дозволяє навчитися точно і формально мислити;
- знання з математики дозволяють програмістові брати участь у цікавих проектах, створенні чогось нового, а не просто використовувати напрацювання попередників;
- за допомогою математики описуються алгоритми.

Де **знання математики обов'язково необхідні** так це при розробці:

- ПЗ, яке пов'язано зі статистичним аналізом;
- завдань математичного моделювання (розробка движків ігор), 3D-моделюванні;
- репутаційних систем;
- високопродуктивних бекендів, що працюють у режимі 24/7, обслуговують мільйони онлайн-користувачів, які тримають сотні тисяч постійних з'єднань;
- програм для напрямку Big Data, Data Mining і Machine Learning (хоча прямим конкурентом в цієї галузі є мова Python).

Де, швидше за все, **не буде потрібне знання математики**, так це у прикладнім програмуванні при розробці простих віконних додатків, веб сайтів (на даний момент цей спектр завдань є основним).

1.6 Питання для самоконтролю

1. Що таке програмування?
2. Які складові програмування?
3. За якими параметрами класифікують мови програмування?
4. Які особливості процедурних мов програмування?
5. Які особливості об'єктно-орієнтованих мов програмування?
6. В чому відмінність мов низького рівня від мов високого рівня?
7. Який зв'язок математики і програмування?
8. Чому мова програмування C++ є перспективною?

2 Основи мови С

С - компільована статично-типізована мова програмування загального призначення, розроблена в 1969-1973 роках Деннісом Рітчі в лабораторіях Bell Labs. Спочатку він був розроблений для реалізації операційної системи UNIX, але, згодом, був перенесений на безліч інших платформ.

Мова програмування С зробила істотний вплив на розвиток індустрії програмного забезпечення, а його синтаксис став основою для таких мов програмування, як C++, C#, Java і Objective-C.

Мова програмування С відрізняється мінімалізмом. Автори мови хотіли, щоб програми на ньому швидко компілювалися за допомогою однопрохідного компілятора. Тому зробили так щоб після компіляції кожної елементарної складової програми відповідало невелике число машинних команд, а використання базових елементів мови не задіяв бібліотеку часу виконання.

Код на С можна легко писати на низькому рівні абстракції, тому іноді С називають «універсальним асемблером» або «асемблером високого рівня», що відображає відмінність мов асемблера для різних платформ і єдність стандарту С.

Код С може бути скомпільовано без будь-яких змін майже на будь-якої моделі комп'ютера. Найчастіше мова програмування С називають мовою середнього рівня або навіть низького рівня, з огляду на те, як близько він працює до реальних пристроїв.

Історія стандартів мови С:

- 1972 - народження мови програмування
- 1978 - K&R C (додано робота з структурами, типами long int, unsigned int, сумісних операторів)
- 1989 - ANSI C (C89)
- 1990 - ISO C (співпадає з C89)
- 1999 - C99
- 2011 - C11
- 2017 - C17 (C11 з виправленням вразливостей, bug-fixing)
- 2018 - C18 (співпадає з C17)

2.1 Алфавіт мови

У тексті на будь-якій природній мові можна виділити чотири основні елементи: символи, слова, словосполучення, речення. Подібні елементи містить і алгоритмічна мова, тільки слова називають лексемами (елементарними конструкціями), словосполучення – виразами, а речення – операторами. Лексеми утворюються із символів, вирази – з лексем і символів, а оператори – із символів, виразів і лексем.

В алфавіт мови С входять:

- великі (прописні) і малі (рядкові) літери латинського алфавіту;

- цифри;
- спеціальні символи: ” , { } | [ ] ( ) + – % \ ; ’ . : ? < = > _ ! & * # ~ ^
- пробільні символи (“загальні пробільні символи”), що використовуються для відокремлення лексем (наприклад, пробіл, табуляція, перехід на нову строку). Компілятор ігнорує пусті строки, “зайві” пробіли, а також знаки табуляції. Винятком є випадок, коли ці символи входять до складу строки. При розумному використанні пробілів підвищується читабельність програми. Наприклад, можна (навіть треба) залишати пусті рядки між великими блоками коду в тому випадку, якщо весь код внутрі блок взаємозв’язаний.
- інші символи: використовуються в коментарях до тексту програми

Алфавіт мови С служить для побудови слів, які називаються лексемами.

Лексеми – одиниці тексту програми, які при компіляції сприймаються як єдине ціле і за змістом не можуть бути розділені на більш дрібні елементи. Розрізняють п’ять типів лексем:

- ідентифікатори;
- ключові слова;
- знаки (символи) операцій;
- літерали;
- роздільники.

2.2 Ідентифікатори і ключові слова

Ідентифікатор – це ім’я програмного об’єкта. Це може бути назва змінної, константи, функції, тощо.

Існує декілька угод про правила створення імен:

- угорська нотація: кожне слово, що входить до складу ідентифікатора, починається з прописної букви, а на початку ставиться префікс, відповідний до типу величини, наприклад, iMaxLength (ціле), fAverageValue (дробове);
- стиль Camel (верблюдица) – перша буква рядкова, інші слова, які є частинами ідентифікатора, починаються із прописної букви, наприклад, maxLength, averageValue;
- ще один стиль пропонує розділяти слова, що становлять ім’я, знаками підкреслення: max_length, average_value, number_of_computer.

Не зважаючи на тип використаної угоди, імена ідентифікаторів повинні бути інформативними, наприклад: sum, average, count, замість s, a, c.

З особливостями, рекомендаціями щодо використання, а також додатковими ресурсами по іменуванню ідентифікаторів, можна ознайомитись в розділі, присвяченому правилам кодування (Code-convention).

Довжина ідентифікатора по стандарту не обмежена, але деякі компілятори та компоновщики накладають на неї обмеження. Ідентифікатор створюється на етапі оголошення змінної, функції, типу і т.п., після цього його можна

використовувати в операторах програми.

При виборі ідентифікатора необхідно враховувати наступне:

- ідентифікатор не повинен збігатися із ключовими словами і іменами використовуваних стандартних об'єктів мови;
- ідентифікатори не повинні повторюватися внутрі області видимості. При цьому, слід враховувати той факт, що імена ідентифікаторів в мові С регістро-залежні, тобто змінна з іменем SIZE та size – дві відокремлені змінні;
- не рекомендується починати ідентифікатори із символу підкреслення, оскільки вони можуть збігтися з іменами системних функцій або змінних, і, крім того, це знижує мобільність програми;
- ідентифікатор не може починатися з цифри;
- пробіли усередині імен не допускаються;
- для поліпшення читаності програми слід давати об'єктам осмислені імена;
- ідентифікатор складається тільки з букв (бажано тільки латинських), цифр та знаку підкреслення;
- на ідентифікатори, використовувані для визначення зовнішніх змінних, накладає обмеження компоновщик (використання різних компоновщиків або версій компоновщика накладає різні вимоги на імена зовнішніх змінних).

Ключові слова – це зарезервовані ідентифікатори, які мають спеціальне значення для компілятора. Їх можна використовувати тільки в тому значенні, у якім вони визначені. Ключові слова діляться на:

- специфікатори типів: char, double, enum, float, int, long, short struct, signed, union, unsigned, void, typedef;
- кваліфікатори типів: const, volatile;
- кваліфікатори класів пам'яті: auto, extern, register, static;
- оператори мови та ідентифікатори спеціального призначення: break, continue, do, for, goto, if, return, switch, while; default, case, else, sizeof;

2.3 Основні типи даних та їх модифікації

Основна мета будь-якої програми полягає в обробці даних. Дані різного типу зберігаються і обробляються по-різному. У будь-якій алгоритмічній мові кожна константа, змінна, результат обчислення виразу або функції повинні мати певний тип. Тип даних визначає:

- внутрішнє подання даних у пам'яті комп'ютера;
- множину значень, які можуть приймати величини цього типу;
- операції та функції, які можна застосовувати до величин цього типу.

Виходячи із цих характеристик, програміст вибирає тип кожної величини, використовуваної в програмі для подання реальних об'єктів. Обов'язковий опис типу дозволяє компіляторові виконувати перевірку допустимості різних конструкцій програми. Від типу величини залежать і машинні команди, які

будуть використовуватися для обробки даних.

Усі типи даних загального значення в мові С зберігаються в пам'яті у вигляді нулів та одиниць. Тому, можна сказати, що використовується лише один тип даних - чисельний. Але це було б недостатньо при реалізації сучасних програм. Тому, виділяють декілька логічних типів даних. До таких "класичних" типів даних відносять символьний тип, цілочисельний тип, речовинний тип. До некласичних типів загального призначення мови С можна навести ще: широкий символьний тип (wchar_t) та булевий тип (bool).

2.3.1 Символьний тип

Використовуються для відображення окремих знаків, що мають індивідуальні внутрішні коди. Кожний символ – це лексема, яка складається із зображення символу і обмежуючих апострофів. Наприклад: 'A', 'a', 'B', '8', '0', '+', ';' і т. д. Детально символьні константи будуть розглядатися при використанні введення-виведення даних.

Зверніть увагу. Будь-який символ в мові С має асоційований код у відповідності з таблицею ASCII (див. https://en.wikipedia.org/wiki/ASCII), тому всі символи в пам'яті представлені у вигляді числових констант.

Величини типу char застосовуються також для зберігання цілих чисел, що не перевищують границі зазначених діапазонів.

2.3.2 Цілочисельний тип

Цілочисельні константи відображають цілі (Не дробові) числа і в мові С діляться на 3 категорії: десяткові, шістнадцяткові і восьмеричні. Основа визначається префіксом в запису константи.

- Для десяткових констант префікс не використовується. Десяткові цілі визначені як послідовності десяткових цифр, починаються не з нуля (якщо це не число нуль). Наприклад, 125, -1, 10, 100500.
- Для вісімкових констант послідовність цифр починається з 0 і не містить десяткових цифр старше 7. Наприклад, 020 - вісімкове подання десяткового цілого 16, а число 090 - є числом 90 в десятирічне системі числення, тому що, незважаючи на лідируючий нуль, в числі присутні цифри старше 7, а значить вісімковий це число не може бути.
- Для шістнадцяткових констант послідовність цифр починається з «0x» (або «0X»), після яких можуть міститися цифрах від 0 до 9, а також символи A, B, C, D, E, F. Наприклад, 0x20 - має значення 32, а число 0xFF - 255

2.3.3 Дійсні числа

Дійсні числа (числа з плаваючою точкою) складаються з наступних частин:

- ціла частина;

- десяткова крапка. Слід запам'ятати, що на відміну від математичної записи, ціла і дробова частина відділяються «крапкою», а не «комою»;
- дрібна частина

Зверніть увагу. Для ідентифікацій наукового формату, після дробової частини, може розташовуватися: ознака показника «e» або «E» та показник десяткової ступеня.

При запису чисел із плаваючою крапкою можуть опускатися ціла або дробова частина (але не одночасно); десяткова крапка або символ експоненти з показником ступеня (але не одночасно).

Приклади констант із плаваючою крапкою:

- *3.14159* ;
- *55.* (ідентично числу 55.0);
- *.01* (ідентично числу 0.01);
- *5.5e2* (ідентично числу $5.5 * 10^2 = 550$);
- *5.5e-1* (ідентично числу $5.5 * 10^{-1} = 0.55$)

2.3.4 Розміри типів даних

Код програми на мові Сі передбачає, що кожна константа, введена в програмі, займає в комп'ютері деякий ділянку пам'яті. Розміри цієї ділянки пам'яті і інтерпретація його вмісту визначаються типом відповідної константи.

Вважається, що тип char, символьний тип даних, відрізняється постійністю розміру і, відповідно, граничних розмірів. На всіх платформах він займає 1 байт.

Для численних типів даних все складніше. Їх розміри, а, отже, і граничні значення залежать від багатьох факторів, таких як архітектура комп'ютера (бітность), тип операційної системи і т.д. Базовим цілочисельним типом даних є тип int. Від англійського слова integer позначає ціле. У сучасних системах займає 4 байта. Тип short int, з англійської означає «короткий», займає в два рази менше пам'яті. Винятком з правила є тип long, який мав би бути в 2 рази більше (8 байт), але в сучасних 32-розрядних системах не має місце.

Для чисел з плаваючою точкою (дійсних чисел) базовим типом даних є float. Від англійського слова «плавати», дане слово символізує опис «плаваюча точка», що говорить, що даний тип є речовим. Прямим «модифікатором» даного типу є тип double, який в 2 рази більше (double - подвійний) типу float.

Слід звернути увагу на те, що при розробці для платформи x64, розмір типу long буде не 4 байта, як зазначено в таблиці, а 8 (також як і у long long). Надалі будуть описані засоби визначення розміру заданого типу даних за допомогою оператора sizeof, а також визначення максимального і мінімального значення за допомогою бібліотеки макровизначень з заголовки `limits.h`.

Table 1 – Типи даних мови C (платформа x32)

Тип даних	Розмір, байт	Min значення	Max значення
char	1	-128	127
short int	2	-32768	32767
int	4	-217483648	217483647
long (long int)	4	$-2 * 10^9$	$2 * 10^9$
long long (long long int)	8	$-9 * 10^{19}$	$9 * 10^{19}$
float	4	$-1 * 10^{38}$	$1 * 10^{38}$
double	8	$-2 * 10^{308}$	$2 * 10^{308}$
long double	16	$-1 * 10^{4932}$	$1 * 10^{4932}$

Особливу увагу слід приділити речовим типам даних. Якщо для цілочисельних типів даних розмір займаної пам'яті впливає тільки на діапазон значень, яке може приймати число цього типу, то для речовинних типів даних уводиться ще один критерій – кількість цифр після коми або точність числа. Точність числа також залежить від архітектури і платформи, на якій виконується розробка:

- для типу float – по специфікації IEEE 754 – 7 символів, але може варіюватися залежно від реалізації від 6 до 9 знаків після коми (у принципі, для повсякденного життя цього завжди достатньо),
- для типу double – по специфікації IEEE 754 – 16 символів, але може варіюватися залежно від реалізації від 16 до 18 знаків після коми (збільшена точність потрібна для математичних розрахунків і при моделюванні процесів),
- для типу long double – від 18 до 21 знаків після коми (специфікація для даного типу даних відсутня).

Для більш детальної інформації про прилади дійсних чисел і їх особливостей, рекомендує звернутися до ресурсів:

- https://en.wikipedia.org/wiki/IEEE_754-1985
- https://en.wikipedia.org/wiki/Single-precision_floating-point_format

2.3.5 Беззнакові числа

Як видно з таблиці, всі типи даних можуть містити в собі як позитивні, так і негативні числа. Іноді з точки зору економії місця доцільно позбутися негативної половини на користь розширення діапазону позитивних чисел. Наприклад, позбутися в типі short int негативною половини, щоб можна було б в нього записати число 45000. Щоб вирішити цю «проблему» необхідно використовувати "беззнаковий" тип даних, який існує для кожного знакового типу. Даний тип даних перед ім'ям типу має ключове слово unsigned, яке говорить само за себе і позначає, що тип беззнаковий. Наприклад, unsigned int, unsigned long, unsigned double. Варто відзначити, що, за замовчуванням

при відсутності в якості префікса ключового слова unsigned будь цілий тип вважається знаковим (signed). Таким чином, вживання спільно зі службовими словами char, short, int, long префікса signed зайве. Припустимо окреме використання позначень (специфікаторів) «знаковості». При цьому signed еквівалентно signed int; unsigned еквівалентно unsigned int. Різниця між цими двома типами - в правилах інтерпретації старшого біта внутрішнього уявлення. Специфікатор signed вимагає, щоб старший біт внутрішнього уявлення сприймався як знаковий; unsigned означає, що старший біт внутрішнього уявлення входить в код наведеного числового значення, яке вважається в цьому випадку беззнаковим.

2.3.6 Суфікси типів даних

За замовчуванням, для всі цілочисельні константи (наприклад, 123) застосовується тип int, а для речовинних (наприклад, 3.14) – тип double, навіть незважаючи на те, що використовуваний тип надлишковий або, навпаки, недостатній (наприклад, для числа 5000000000). Іноді дане поведінка не влаштовує програміста, і щоб явно вказати тип цілочисельної константи, використовуються суфікси, що ставляться відразу після числа:

- F (або f) – float (для речовинних). Наприклад, 3.14159f – уже типу float, а не double.
- U (або u) – unsigned (для цілих), 123U – беззнакова константа типу unsigned int
- L (або l) – long (для цілих і речовинних). Наприклад, 3.14L – уже типу long double, а не просто double; число 123L – типу long.
- UL (або ul) – unsigned long (для цілих).
- LL (або ll) – long long int (для цілих).

2.4 Змінні та іменовані константи

Одним з основних понять мови С є об’єкт – іменована область пам’яті. Окремий випадок об’єкта – змінна. Відмінна риса змінної полягає в можливості зв’язувати з її іменем різні значення, сукупність яких визначається типом змінної. Поняття «змінна» бере початок від англійського слова variable, що дає розуміння того, що значення області пам’яті можна міняти. При завданні значення змінної у відповідну їй область пам’яті міститься код цього значення. Доступ до значення змінної забезпечує її ім’я, а доступ до ділянки пам’яті можливий тільки по його адресі.

Операції, які можливі над змінними:

- оголошення;
- ініціалізація;
- сполучене (разом) оголошення та ініціалізація;
- використання (присвоювання значення, одержання значення).

2.4.1 Оголошення (декларація) змінних

Кожна змінна перед її використанням у програмі повинна бути визначена, тобто для змінної повинна бути виділена пам'ять. Розмір ділянки пам'яті, яка виділяється для змінної, і інтерпретація вмісту залежать від типу, що зазначений у визначенні змінної

Найпростіша форма визначення змінних така:

```
data_type variables_list,
```

де:

- data_type – один із усіх можливих типів даних мови С;
- variables_list – один або кілька ідентифікаторів – імен змінних, які будуть мати зазначений тип.

Щоб оголосити змінну, необхідно привласнити їй ім'я – ідентифікатор. Щоб створити припустимий ідентифікатор, необхідно додержуватися правил створення ідентифікаторів (див. підрозділ "Ідентифікатори і ключові слова").

Приклади оголошення змінних. Слід звернути увагу на те, що наприкінці рядка, де оголошується змінна обов'язково ставиться символ «крапка з комою» (див. підрозділ "Термінальна інструкція"):

```
int height;

char letter;

float lengthA, lengthB;

unsigned int countOfApples;
```

2.4.2 Ініціалізація змінних

Відповідно до синтаксису мови змінні після визначення за замовчуванням мають невизначені значення. Сподіватися на те, що вони дорівнюють, наприклад, 0, не можна. Однак змінним можна привласнювати початкові значення, явно вказуючи їх у визначеннях:

```
type variable_name = initial_value;
```

Це називається ініціалізацією. На відміну від присвоювання, яке здійснюється в процесі виконання програми, ініціалізація виконується при присвоюванні первісного значення змінної. Приклади оголошення змінних з ініціалізацією:

```
float pi = 3.1415f;

int countOfAngles = 4;

float angleOX = 53.5f, angleOY = 33.2f;
```

2.4.3 Константи

У мові С, крім змінних, які можуть міняти своє значення, можуть бути визначені константи, що мають фіксовані значення.

У якості констант використовуються дані, які ні за яких умов не припускають зміни свого значення. Наприклад, число $\pi = 3.14$. Дане число загальновідоме,

має фіксоване значення. У якості імен констант використовуються довільно обирані програмістом ідентифікатори, що не збігаються із ключовими словами і з іншими іменами об'єктів.

У якості другого прикладу можна привести набір вхідних даних, з якими слід працювати надалі і які не слід змінювати в ході життєвого циклу додатка. Наприклад, інформація про значення ключа реєстру Windows, де зберігається конфігурація додатка.

Традиційно прийнято, що для позначень констант вибирають ідентифікатори з великих букв латинського алфавіту і символів підкреслення. Така угода дозволяє при перегляді великого тексту програми мовою С легко відрізняти імена змінних від назв констант.

У мові С існує 2 можливості оголошення констант:

- за допомогою ключового слова const. Слід пам'ятати, що дане ключове слово не є рідним для класичного С, бо воно заємствовано з мови С++ та його використання в pure-C додатків є моветон:

```
const type constant_name = constant_value;
```

де const – квалификатор типу, який вказує не те, що обумовлений об'єкт має постійне значення, тобто доступний тільки для читання; type – один з типів об'єктів; constant_name – ідентифікатор; constant_value повинне відповідати її типу. Приклади:

```
const double PI = 3.1415;

const long MAX_SIZE = 1000;
```

- за допомогою конструкції (директиви препроцесора) #define

```
#define constant_name constant_value
```

Наприклад,

```
#define PI 3.1415

#define MAX_SIZE 1000
```

Слід звернути увагу на відсутність символа «крапка з комою» наприкінці директиви, а також на відсутність знака присвоювання і типу даних.

Незважаючи на те, що в результаті буде одержано з візуальної точки зору однакові константи, є ряд особливостей:

- директива #define на етапі компіляції шукає всі використовувані ідентифікатори (значення і коментарі ігноруються) і заміняє їх вказаним значенням. Детальніше про це буде сказано в розділі «Передпроцесорна обробка». На поточному етапі необхідно лише розуміти, що константи, описані через #define не мають комірки пам'яті.
- при роботі із const помилки, які пов'язані з неправильною роботою, більш лаконічні і зручні для сприйняття:
  - Непроініціалізована константа:

```
const int value2;  // compilation error. need to be initialized
```

При використанні #define помилки будуть показуватися в рядках, де використовується «порожня» константа, що призводить до додаткових тимчасових витрат на обчислення причини помилки (адже в тому рядку

може бути використано багато констант, і не завжди легко зрозуміти де саме помилка).

– Переініціалізація константи:

```
const int value = 100500; value = 123; // assignment of read-only variable 'x'
```

При використанні #define можливі два варіанти розвитку подій:

•

```
#define X 5
#define X 10
```

В даному випадку помилки компіляції не буде і значення може бути змінене

•

```
#define X 5
X= 10;
```

По факту, в даному випадку виконується наступна дія: $5 = 10$, що є логічно невірним. Тому буде видана така помилка компіляції `'=' : left operand must be l-value` яка є малоінформативною.

• #define не перевіряє відповідність типів, Наприклад не буде помилки, якщо ми зробимо таку змінну “Кількість столів”, що логічно повинна бути цілочисельна:

```
#define TABLES 4.5
```

• #define є чисто С-механізмом, а const – варіант перенесений з С++

Крім ідентифікаторів, що визначають змінні і константи, при розробці програмного забезпечення зустрічаються також такі ідентифікатори, що зарезервовані в мові, які не можна використовувати в якості вільно обраних програмістом імен. Такі ідентифікатори називають службовими словами. Службові слова визначають типи даних, класи пам’яті, кваліфікатори типу, модифікатори та оператори. У ході навчання всі ключові слова будуть розглядатися. Слід звернути увагу, що сучасні середовища розробки виділяють ключові слова серед інших ідентифікаторів.

2.5 Оператори і їх пріоритети

Для формування і наступного обчислення виразу використовуються операції. Для зображення однієї операції в більшості випадків використовується кілька символів.

Наявні операції в мові С, згруповані по рангах:

• ::, [ ], ( ), ., ->, ~ (порозрядне НІ), & (одержання адреси), * (розіменування покажчика)
• логічне «НІ»: !
• інкремент/декремент (унарні операції): ++, --
• мультиплікативні оператори: *, /, %,
• адитивні оператори: +, - ,
• оператори бітового зсуву : » , «

- оператори порівняння: <, >, <=, >=, ==, !=
- порозрядні оператори: & (ТА), ^ (XOR), | (АБО)
- логічне «ТА» : &&
- логічне «АБО»: ||
- тернарний оператор: (x == t) ? y : x
- оператори присвоєння: =, +=, -=, /=, *= , &=

За винятком операцій “[ ]”, “( )” и “?:”, усі знаки операцій розпізнаються компілятором як окремі лексеми. Залежно від контексту та сама лексема може позначати різні операції, тобто той самий знак операції може вживатися в різних виразах і по-різному інтерпретуватися залежно від контексту. Наприклад, бінарна операція & – це порозрядна кон’юнкція, а унарна операція & – це операція одержання адреси.

Зауваження: якщо є невпевненість у пріоритетах операцій – краще перестрахуватися і використовувати оператор ( ) для визначення пріоритетів.

РЕМАРКА: Багато хто вважають, що поняття оператор і операція – одне і теж саме. Насправді ці поняття схожі, але на думку автора мають одну відмінність: операція – дія, яка виконується над операндами (наприклад, сума, множення, модуль числа), а оператор – символ, який відповідає за виконання операції (наприклад, +, *, %).

Призначення всіх операторів буде розкриватися по ходу вивчення курсу.

Опис і приклади використання базових операторів.

Перед описом операцій слід виділити таке поняття як «ліворуч припустиме значення». Вирази, що мають ім’я і модифікуються (L-value, left value, l-значення, ліворуч припустимий вираз) походять від пояснення дії операції присвоювання E = D, у якій операнд E ліворуч від знака операції присвоювання може бути таким, що тільки модифікується l-значенням. Прикладом, що модифікується l-значенням служить ім’я змінної, для якої виділена пам’ять. Таким чином, l-значення це – посилання на область пам’яті, значення якої може змінюватися.

2.5.1 Інкремент та декремент

- ++ – інкремент (збільшення на одиницю, ідентично виразу +1), унарна операція;
- -- – зменшення на одиницю (декремент або автозменшення), унарна операція.

Операнд для операції ++ (і для операції --) не може бути константою або довільним виразом. Наприклад такі записи *++5* або *84++* будуть помилковими. *++(j+k)* – також помилковий запис. Операндами унарних операцій ++ й -- мають бути завжди «ліворуч припустимі значення».

Наприклад, якщо i = 5; то вираз i++ в результаті приведе до того, що значення змінної і стало = 6.

Інкремент, і декремент мають дві форми:

- префіксна операція – зміна на одиницю значення операнда до його використання, наприклад, $++k$;
- постфіксна операція – зміна на одиницю значення операнда після його використання, наприклад, $k++$;

Якщо вираз складається тільки з операції інкремента / декремента, то різниці між даними формами не буде. Різниця має місце, коли використовується вираз, у якім використовуються трохи математичні операції, у тому числі може бути інременти та декременти. Наприклад, нехай $k = 3, j = 5$.

- y = k++; після виконання оператора присвоювання буде таке: y = 3, k = 4. Змінна k збільшилася після того, як присвоювалася результуючій змінній.
- y = ++k; після виконання оператора присвоювання y = 4, k = 4. Змінна k спочатку збільшилася, а потім присвоїлася результуючій змінній.
- y = k++ + 3 + j--; після виконання оператора присвоювання будуть такі значення: y = 11, k = 4, j = 4. Операції постінкремента та постдекремента виконалися після того, як сформувалася змінна y = 3 + 3+ 5, де перша 3 – старе значення змінної k, а 5 – старе значення змінної j.
- y = --j + j++; після виконання оператора присвоювання y = 8, j = 5. У результаті дії, значення j спочатку зменшилося на 1 (стало дорівнювати 4), далі зложився із собою (=4). Після присвоєння результату став дорівнювати 5.
- завдання про «п'ять плюсів». $x = k++ + ++k$. Тут 5 плюсів діляться на 3 групи:
  - перша група має відношення до операції постінкремента;
  - друга група операція суми;
  - третя група - операція передінремента Після виконання виразу $x = 8$, а $k = 5$. **Пояснення**: початкове значення змінної k дорівнює 3; за рахунок преінкремента збільшується на один, складується із собою (4+4) і за рахунок постінкремента збільшується ще на одиницю.

2.5.2 Математичні операції

Операції додавання, віднімання та множення відомі зі школи, і ніяких особливостей не мають. Особливості представляють операції ділення та залишку від ділення (ділення по модулю).

- / – ділення операндів арифметичного типу. Слід зазначити, що результат ділення буде завжди округлятися до типу, найбільшого із двох. Так при цілочисельних операндах (наприклад, обоє типу int) абсолютне значення результату округляється до цілого. Наприклад, 18/5 = 3, –20/3 = –6; Для того, щоб результат став речовинним числом, необхідне ділене та/або дільник привести до речовинного типу, наприклад: 18.0 / 5, 1/5.0, 3.1 / 5.3. Зміна типів даних не завжди можлива, тому для розв'язку даної проблеми використовують також явне приведення типів (див. далі).
- % – одержання залишку від ділення цілочисельних операндів. З

математичної точки зору залишок по модулю M числа K вираховується по формулі:

```
M – (trunc(M / K)) * K
```

де trunc – процес відкидання дробової частини.

Наприклад:

```
17 % 4 = 17 - (trunc(17 / 4)) * 4 = 17 – 4 * 4 = 17 – 16 = 1
```

В операції залишку від ділення є одна особливість: результат залишку від ділення завжди буде належати діапазону від 0 до значення модуля M – 1. У випадку з вищенаведеним прикладом – у діапазоні від 0 до 3 (включно). Доведемо це емпіричним шляхом - створимо таблицю залишків від ділення на число 4:

```
0 % 4 = 0   1 % 4 = 1   2 % 4 = 2   3 % 4 = 3
4 % 4 = 0   5 % 4 = 1   6 % 4 = 2   7 % 4 = 3
```

2.5.3 Оператори бітового зсуву

- < < – зсув вліво бітового представлення значення лівого цілочисельного операнда на кількість розрядів, що дорівнює значенню правого цілочисельного операнда. При цьому, молодші розряди заповнюються нулями;
- > > – зсув вправо бітового представлення значення лівого цілочисельного операнда на кількість розрядів, що дорівнює значенню правого цілочисельного операнда. При цьому, розряди, які вийшли за розрядну сітку, губляться і надалі не враховуються.

Слід зазначити факт відсутності стандарту на правило заповнення лівих розрядів, які звільнюються. У стандарті мови сказано, що коли лівий операнд є ціле значення з негативним знаком, то при зсуві вправо заповнення вивільнюваних лівих розрядів визначається реалізацією. Тут можливі два варіанти:

- звільнювані розряди заповнюються значеннями знакового розряду (арифметичний зсув вправо)
- звільнювані ліворуч розряди заповнюються нулями (логічний зсув вправо)

Приклади результатів виконання операцій зсуву:

```
4<<2 = 00000100b << 2 = 00010000b = 16;
5>>1 = 00000101b >> 1 = 00000010b = 2;
5>>3 = 00000101b >> 3 = 00000000b = 0;
4<<4 = 00000100b << 2 = 01000000b = 64;
```

З наведених прикладів можна виявити таку закономірність:

- зсув на 1 розряд вліво співпадає з множенням на 2. У загальному випадку, зсув на K розрядів вліво співпадає з множенням на 2^K;
- зсув на 1 розряд вправо співпадає з діленням на 2, при цьому дробова частина відкидається. У загальному випадку, зсув на K розрядів вправо співпадає з діленням на 2^K

2.5.4 Порозрядні (по-бітові) оператори

- & – порозрядна кон'юнкція (ТА) бітових представлень значень цілочисельних операндів;
- | – порозрядна диз'юнкція (АБО) бітових представлень значень цілочисельних операндів;
- ^ – порозрядне виключне АБО бітових представлень значень цілочисельних операндів;
- ~ – порозрядна інверсія внутрішнього двійкового коду цілочисельного аргументу – побітове заперечення. При цьому кожний біт числа перетворюється згідно з таблицею істинності заперечення (НІ).

Порозрядні операції дозволяють конструювати вирази, у яких обробка операндів виконується на бітовому рівні (порозрядно). При цьому при роботі із двохопередними операціями, двійкове представлення одного операнда записується під двійковим представленням другого операнда з вирівнюванням записів по лівому краю (так само, як виконується операція додавання двох чисел у стовпчик) і кожна пара бітів (один біт верхнього числа і відповідний один біт нижнього числа) обробляються згідно з таблицею істинності. Результат обробки кожної пари бітів не впливає на результати обробки сусідніх пар бітів.

Table 2 – Таблиця прикладів обчислення бітових операцій

Обчислення в 10-річному форматі	Обчислення в аналогічному двійковому форматі	Результат (10-річний)	Результат (двійковий)
5 & 7	0101 & 0111	5	0101
5 & 6	0101 & 0110	4	0100
10 | 6	1010 | 0110	14	1110
7 ^ 13	0111 ^ 1101	10	1010
unsigned char a = 25;	~ 00011001	230	11100110

Table 3 – Таблиця істинності кон'юнкції (ТА), диз'юнкції (АБО), що виключає АБО (XOR):

x	y	Результат ТА	Результат АБО	Результат XOR
0 («неправда»)	0 («неправда»)	0	0	0
0 («неправда»)	1 («правда»)	0	1	1
1 («правда»)	0 («неправда»)	0	1	1
1 («правда»)	1 («правда»)	1	1	0

Table 4 – Таблиця істинності заперечення (НІ):

x	Результат
0 («неправда»)	1
1 («правда»)	0

2.5.5 Операції порівняння

Операції порівняння повертають результат у вигляді «правда» або «неправда». Операнди у виразах повинні бути одного типу (не можна порівнювати, наприклад, рядок з речовинним числом).

До операцій порівняння відносять:

- < (менше), > (більше);
- <= (менше або рівно), >= (більше або рівно). Слід звернути увагу, що оператор «=» пишеться після знака більше/менше;
- == (порівняння на рівність);
- != (порівняння на нерівність).

2.5.6 Логічні операції

Даний тип операцій застосовується тільки до логічних типів даних. Оскільки у мові С логічного типу даних не існує, замість нього використовуються цілочисельні типи даних, при цьому діє правило, що 0 – «неправда», а 1 – «правда». Процес виконання логічних операцій схожий з побітовими, за винятком того, що в логічних операціях обробляється тільки один, молодший біт, який може бути або 0 або 1.

Логічні операції ділять на:

- && - кон'юнкція (ТА) арифметичних операндів або відносин. Цілочисельний результат 0 (неправда) або 1 (правда);
- || - диз'юнкція (АБО) арифметичних операндів або відносин. Цілочисельний результат 0 (неправда) або 1 (правда).
- ! - унарна операція заперечення (прикладом є лише таблиця істинності для заперечення).

Приклади роботи операцій порівняння та логічних операцій:

```
2 < 3, результат 1;

2 > 3, результат 0;

5 != 6, результат 1;

10 > 5 && 10 < 20, результат 1;

10 > 15 || 10 > 20, результат 0.
```

2.5.7 Тернарна операція

На відміну від унарних (один операнд) і бінарних (два операнди) операцій тернарна операція використовується із трьома операндами. У запису умовної операції застосовуються два символи ‘?’ та ‘:’ і три вирази-операнда:

```
ввираз_1 ? вираз_ 2 : вираз_3
```

Першим обчислюється значення вираз_1. Якщо воно «правда», тобто не дорівнює нулю, то обчислюється значення вираз_2, яке стає результатом. Якщо при обчисленні вираз_1 вийде 0, то в якості результату береться значення вираз_3.

Класичний приклад визначення модуля (абсолютну величину) числа:

```
x < 0 ? -x : x;
```

2.5.8 Операції присвоювання

Символ «=» у мові С позначає бінарну операцію, у якій у виразу має бути два операнди: лівий (вираз, що модифікується, як правило – змінна) і правий (як правило вираз).

Наприклад, якщо z – ім’я змінної, то

```
z = 2.3 + 5.1;
```

це вираз зі значенням 7.4. Одночасно це значення привласнюється і змінній z. У якості лівого операнда в операціях присвоювання може використовуватися тільки ім’я, що модифікується (l-значення), тобто посилання на деяку іменовану область пам’яті, значення якої можна змінити.

Тип і значення виразу з операцією присвоювання визначаються значенням виразу, поміщеного праворуч від знака «=». Однак цей тип може не збігатися з типом змінної з лівої частини виразу. У цьому випадку при визначенні значення змінної виконується перетворення (приведення) типів.

Суміщені операції присвоювання створені для того, щоб спростити код і життя програмістові. Вони дозволяють заміняти операції виду:

```
var1 = var1 OPERATION value
```

на конструкцію виду:

```
var1 OPERATION = value
```

де :

- OPERATION – одна з нижче перерахованих операцій:
  - математичні операції: *, /, %, +, -
  - операції зсуву: «, »
  - побітові операції: &, |, ^
- value – другий операнд.

Наприклад, конструкцію $x = x + 10$; можна замінити на $x+ = 10$;
А конструкцію $y = y/(x * 2)$ можна замінити на $y/ = (x * 2)$.

2.6 Приведення типів

Розглядаючи операцію ділення, було відзначено, що при діленні двох цілих операндів результат буде цілим. Для одержання речовинного результату потрібно виконувати ділення не цілих, а речовинних операндів. Якщо операндами є безіменні константи, то замінити цілу константу на речовинну зовсім не складно. У тому випадку, коли операндом є іменована константа, змінна або вираз в дужках, необхідно для розв'язку завдання використовувати операцію явного приведення (перетворення) типу.

Наприклад, є такий набір визначень і операторів присвоювання

```
int n = 5, k = 2;
double d;
int m;
d = (double) n / (double) k;
m = n / k;
```

У цьому фрагменті змінна d буде мати значення 3.5 типу $double$, а значенням змінної m стане ціле число 3.

Операція ділення – це тільки одна з бінарних операцій. Майже для кожної з бінарних операцій операнди можуть мати різні типи. Однак не завжди програміст повинен у явному вигляді вказувати перетворення типів.

Якщо в бінарній операції операнди мають різні типи (а відповідно до синтаксису виразу має бути один тип), то компілятор виконує перетворення типів автоматично, тобто приводить обидва операнди до одного типу. Наприклад, для тих же змінних значення виразу $d + k$ буде мати тип $double$ за рахунок неявного перетворення, яке виконується автоматично без вказівки програміста. Далі розглянуті правила, по яких такі приведення виконуються.

2.6.1 Правила перетворення типів

При обчисленні виразів деякі операції вимагають, щоб операнди мали відповідний тип, а якщо вимоги до типу не виконані, примусово викликають виконання потрібних перетворень. У мові С присвоювання є бінарною операцією, тому правило щодо перетворення типів має відношення до усіх форм присвоювання. Однак при присвоюваннях значення виразу з правої частини завжди приводиться до типу змінної з лівої частини, незалежно від співвідношення цих типів.

При виконанні перетворення операндів в арифметичних виразах, **без явного перетворення** відбувається перетворення типів до найбільшого типу даних. Старшинство визначається наступною послідовністю:

- long double
- double
- float
- unsigned long int

- long
- unsigned int
- int

2.6.2 Операція явного перетворення типу

Операція перетворення (приведення) типу має наступний формат:

```
(type_name) operand
```

Такий вираз дозволяє перетворювати значення операнда до заданого типу. У якості операнда використовується унарний вираз, який в найпростішому випадку може бути змінною, константою або будь-яким виразом, укладеним у круглі дужки. Наприклад, перетворення (long) 8 (внутрішнє представлення результату має довжину 4 байти) і (char) 8 (внутрішнє представлення результату має довжину 1 байт) змінюють довжину внутрішнього представлення цілих констант, не міняючи їх значень. У цих перетвореннях константа не міняла значення і залишалася цілочисельною. Однак можливі більш глибокі перетворення. Наприклад (long double)6 або (float)4 не тільки змінюють довжину константи, але і структуру її внутрішнього представлення.

Приклади:

```
long i = 12L; /* variable declaraction and initialization */
float brig; /* variable declaraction */
brig = (float)i; /* Explicit type cast - `brig` equals to 12L, casted to `float` type*/
```

В цьому прикладі brig одержує значення 12L, перетворене до типу float.

При явнім перетворенні типів може відбуватися втрата даних.

Наприклад, коли змінну речовинного типу перетворюють до змінної цілого типу (губиться дробова частина), або коли змінну типу int зі значення 40000000 перетворюють до типу short (у цьому випадку старші біти відкидаються і на виході буде число 40000000 % 65535 = 23650.

Нижче наведені «безпечні» арифметичні перетворення (позначені стрілками), що гарантують збереження точності і незмінність значення числа:

- signed char (-128..127) -> short -> int -> long
- float -> double -> long double
- unsigned char -> unsigned short -> unsigned int -> unsigned long

Явне перетворення типів корисно, коли діляться дві змінні цілого типу і при цьому треба зберегти дробову частину. Як говорилося раніше, при діленні двох цілочисельних змінних результат завжди буде цілим числом. Для збереження дробової частини результату треба перетворити один з операндів до речовинного типу. Наприклад:

```
int sum = 16;
int count = 3;
float avg3 = sum / (float) count;
```

2.6.3 Правила явного перетворення типів

Перетворення цілих типів зі знаком

- Перетворення цілого зі знаком у число плаваючого типу відбувається без втрати інформації, за винятком випадку перетворення типу long int або unsigned long int до типу float, коли точність часто може бути загублено.
- Ціле зі знаком перетворюється у більш довге ціле зі знаком шляхом розмноження знакового розряду (знаковий розряд – самий старший розряд пам'яті, яка виділена для числа). **Пояснення**: усі додані розряди двійкового числа будуть мати таке ж значення, яке мав знаковий розряд початкового числа: якщо число було позитивним, те це буде 0, якщо негативним – 1. З точки зору програмування, число не зміниться.
- Ціле зі знаком перетворюється у ціле без знака. На першому кроці ціле зі знаком перетвориться у ціле зі знаком відповідно до цільового значення – тобто, того, до якого приводиться (якщо цільовий тип даних більше). На другому кроці у значення, яке було отримано на першому кроці, знак не відкидається, просто всі розряди вважаються числовими (такими, що визначають значення числа), у тому числі і знаковий розряд. Наприклад,

```
short x = -123;
unsigned short y = x; // = 65413
```

Пояснення: у пам'яті число -123 = `1111 1111 1000 0101`, де старший розряд указує на те, що число – негативне. У беззнаковому представленні старший розряд відіграє роль числового розряду 35536 (215). В результаті число буде дорівнювати 65413

- Ціле зі знаком перетворюється у більш коротке ціле зі знаком, із втратою інформації: усі розряди числа, які перебувають вище (або, відповідно – нижче) границі, що визначає максимальний розмір змінної, губляться. Наприклад:

```
int x = -63000;
short y = x;  // = 2536
```

Пояснення: для даних типу *int* виділяється 4 байти. Тож у пам'яті, оскільки x – від'ємне, число −63000 буде записано так: `1111 1111 1111 1111 0000 1001 1110 1000`. Для даних типу *short* виділяється 2 байти. Як наслідок, у пам'яті, що виділена для у типу short може поміститися тільки 4 молодші тетради (2 байта), тож значення змінної у дорівнюватиме `0000 1001 1110 1000` , що є числом 2536.

Перетворення цілих типів без знаку

- Ціле без знака перетворюється у більш коротке ціле без знака або зі знаком шляхом усікання значення числа.
- Ціле без знака перетворюється у більш довге ціле без знака або зі знаком шляхом додавання нулів ліворуч. По факту, число не змінюється.
- Ціле без знака перетворюється у ціле зі знаком того ж розміру. Якщо взяти для прикладу, unsigned short і short – числа в діапазоні 32768-65535 перетворяться в негативні.

- Ціле без знака перетворюється у плаваючий тип. Спочатку воно перетвориться до значення типу signed long, яке потім перетвориться в плаваючий тип.

Перетворення плаваючих типів.

- Величини типу float перетворяться до типу double без зміни значення.
- Величини double перетворяться до float з деякою втратою точності, тобто, кількості знаків після коми. Якщо значення занадто велике для float, то відбувається втрата значності, про що повідомляється під час виконання перетворення.
- При перетворенні величини із плаваючою крапкою до цілих типів вона спочатку перетвориться до типу long (дробова частина плаваючої величини при цьому відкидається), а потім величина типу long перетвориться до необхідного цілого типу. Якщо значення занадто велике для long, то результат перетворення не визначений.

2.7 Структура програми.

Сучасні програми, як правило, дуже великі за розміром, мають складну структуру. Спробувати відразу вивчати всі можливі складові програм не має сенсу бо це – не просте завдання. Курс же передбачає вивчення матеріалу з самого початку. Тому програми будуть писатися спочатку маленькі, з поступовим розширенням їх функціональності.

Майже кожна програма мовою С складається із точки входу, місця звідки починають виконуватися інструкції (команди). Дана точка входу має фіксоване ім’я – **main**. У загальному вигляді структура найпростішої програми виглядає в такий спосіб:

```
int main( )
{
    init_block;
    actions;
    return 0;
}
```

У розділі *init_block* виконується оголошення та ініціалізація констант, вхідних змінних, а також оголошення проміжних і результуючих змінних, з якими необхідно працювати в процесі життєвого циклу додатка. За оголошенням змінних ідуть *actions* - операції, які приводять до розв’язку поставленого завдання.

Надалі структура програми буде розширюватися.

Слід зазначити, що точка входу main має бути одна в програмі. А якщо - ні, то при компіляції програми буде видане повідомлення про помилку.

2.8 Коментарі

Рядки коментарів компілятор ігнорує тому, що вони призначені для людей, а не для машини. Коментарі допомагають іншим програмістам, які будуть читати код, зрозуміти хід виконання програми. Також, коментарі можуть знадобитися і самим розробникам програми. Вони вказують те, як було розв'язано те або інше завдання, тоді як в коді програми це було не очевидно.

Коментарі бувають двох видів:

- однорядкові – позначаються двома слешами (//). Після двох слешів увесь текст до кінця фізичного рядка є коментарем і компілятор його буде ігнорувати. **Зверніть увагу!**. Незважаючи на те, о С дозволяє використовувати даний тип коментарів, не рекомендується його використовувати, бо він не є "рідним" - він запозичений з мови С++.
- коментарі в стилі С – можуть займати частину рядка, один рядок або кілька рядків. Такий коментар починається із символів /*, і закінчується символами */. При цьому текст коментаря може бути в середині рядка, що дозволяє виконати один їх варіантів коментування рядка коду.

Коментарі записуються або перед блоком, для якого пишеться коментар, або в тому ж рядку, що і коментована дія.

У коментарях, рядках і символьних константах можуть використовуватися і інші літери (наприклад, російські букви).

Коментар формується як послідовність знаків (символів), що обмежуються ліворуч знаками /*, а праворуч – знаками */. Наприклад:

```
/* This is commented block */
```

У стандартній мові С коментарі заборонено вкладати один в інший. Тобто запис:

```
/* code-1 /* code-2 */ code-3 */
```

має помилку – code-3» не вважається коментарем.

2.9 Термінальна інструкція

Компілятор С дозволяє розбивати один логічний рядок на кілька фізичних рядків. Для того, щоб ідентифікувати кінець логічного рядка, використовується термінальна інструкція – «;».

Будь-який допустимий вираз, за яким іде «;», сприймається як оператор. Це справедливо і для порожнього виразу, тобто окремий символ «крапка з комою» вважається порожнім оператором.

2.10 Оператор переносу строки

Іноді трапляється ситуація, коли вираз дуже довгий і його необхідно розбити на декілька рядків. Наприклад,

```
X = value1 * 1+ value2 *  + value3 * 3+ value4 * 4;
```

Процес розбивки рядків звичайним переносом рядка приведе до синтаксичних помилок. Щоб правильно розбити рядок на кілька рядків, необхідно використовувати символ «» (зворотний слеш) наприкінці рядка. При цьому, після цього символу не повинно бути ні одного символу (навіть пробілу):

```
X = value1 * 1\
    + value2 * 2\
    + value3 * 3\
    + value4 * 4;
```

2.11 Розробка першої програми

За для закріплення пройденого матеріалу необхідно розв'язати таке завдання: по заданому радіусу і висоті циліндра визначити його об'єм. Написати програму мовою С, при умові, що радіус = 20 мм, висота = 15.5 мм.

Що нам потрібно? Знання математики. Без неї ніяк. Нам необхідна формула обчислення об'єму циліндру: $V = pi * R^2 * h$.

Дії, що необхідні для реалізації програмного коду:

0. Створити файл `main.c` у директорії `src`. Будемо вважати, що поточний проект знаходиться у наступній директорії: `~/dev/project_linear`.
1. Зробимо шаблон нашого коду у файлі `main.c`:

```
int main() {
    // (1)
    // (2)
    // (3)
    return 0;
}
```

2. Далі ми повинні описати вхідні змінні та константи в блоці (1):

```
#define PI 3.14
const int RADIUS = 20;
const float HEIGHT = 15.5f;
```

3. В блоці (2) опишемо результуючу змінну. Враховуючи той факт, що для отримання результуючого значення у нас використовуються дійсні числа, кінцевий результат теж повинен бути дійсним:

```
float result;
```

4. У блоці (3) виконуються обчислення. Для піднесення числа у другий ступінь змінна *radius* буде помножена сама на себе:

```
result = PI * RADIUS * RADIUS * HEIGHT;
```

Разом увесь код має такий вигляд:

```
// Program calculated cylinder volume with height=15.5mm and radius=20mm.
int main() {
    // Pi constant
    #define PI 3.14
    // Radius. Per task it is integer. Will use this type
```

```
    const int RADIUS = 20;
    // Height. Per task it is real (float-pointer) number.
    const float HEIGHT = 15.5f;
    float result;
    result = PI * RADIUS * RADIUS * HEIGHT;
    return 0;
}
```

2.11.1 Компіляція

З інформації інших галузь нам відомо, що код на будь-якої мові програмування (крім машинів кодів) - це деяка оболонка над машиними командами. При цьому, машина (ЕОМ) не розуміє ці команди, хоча вони зручні та розумілі для людини. Для того, щоб перетворити команди мови програмування в машинні команди - існує процес компіляції (на самом ділі - цей процес більш складний та буде розглянуто декілька далі).

Для компіляції програм, написаних на мові С, нам потрібен С-компілятор. Для Linux-подібних систем, в більшості випадків - це GCC (GNU C Compiler). Щоб проінсталювати його, потрібно виконати наступну команду:

```
sudo apt install gcc
```

Наступний запуск команди `gcc --version` повинен виконатись успішно та Ви побачене версію Вашого компілятора.

Зверніть увагу. Замість компілятора gcc рекомендується використовувати його сучасний аналог “clang”. Він має тій же набор команд, що й gcc.

Щоб скомпілювати вашу програму, необхідно виконати наступну команду:

```
cd ~/dev/project_linear
gcc -std=gnu11 -g \
    -Wall -Wextra -Werror -Wformat-security -Wfloat-equal -Wshadow \
    -Wconversion -Wlogical-not-parentheses -Wnull-dereference \
    src/main.c -o ./dist/main.bin
```

Розберемо, що тут коється:

- `gcc` - команда компілятора GCC
- `-std=gnu11` - параметр компілятора, що визначає компіляцію за стандартом C11. Якщо він не буде вказаний - буде проходити класична компіляція за більш старим стандартом C98
- `-g` - параметр компілятора, що визначає додання інформації для відлагодника. У “продакшені” цей параметр убирають:
  - при компіляції в режимі debug (з додатковою інформацією) розмір програми більше ніж при компіляції в режимі release (без додаткової інформації);
  - якщо програма скомпільована в режимі release, то у користувача майже нема можливостей відлагоджувати програму, бо інформація для відлагодження відсутня;

  - при компіляції в режимі release виконується оптимізація програми, тому якщо існує «мертвий» код, або код, дії якого ні до чого не призводять, може бути видалений за рішенням компілятора.

- `-Wextra -Werror -Wformat-security -Wfloat-equal -Wshadow -Wconversion -Wlogical-not-parentheses -Wnull-dereference` - перелік параметрів компілятора, що був визначений емпіричним шляхом, що дозволяє для лабораторних робіт проводити базову перевірку якості коду та визначати вразливі місця.
- `src/main.c` - головний файл з функцією main
- `-o ./dist/main.bin` - розположення а ім'я результуючого (скомпільованого) файлу.

З опису команди ми маємо виявити наступне:

- результуючий файл знаходитимеся в директорії, що не існує. Компілятор не буде створювати її за вас. Тому треба зробити це за вас
- при компіляції файл `./dist/main.o` вже може існувати. У подальшому, в каталозі `./dist/` можуть бути додаткові тимчасові файли, які можуть мати поганий вплив на компіляцію. Щоб уникнути цього впливу, необхідно видаляти цю директорію перед компіляцією.

Таким чином, компіляція нашого проекту має наступний вигляд:

```
cd ~/dev/project_linear
rm -rf ./dist
mkdir ./dist
gcc -std=gnu11 -g \
    -Wall -Wextra -Werror -Wformat-security -Wfloat-equal -Wshadow \
    -Wconversion -Wlogical-not-parentheses -Wnull-dereference \
    src/main.c -o ./dist/main.bin
```

Результат компіляції:

- успішна компіляція без зауважень. Команда виконалась та нічого не вивела. Результуючий файл створився.
- успішна компіляція з зауваженнями. Результуючий файл створився. В результаті компіляції вказується в якому файлі та якій строчці є зауваження. В кінці повідомлення вказується кількість зауважень. В рамках виконання лабораторних робіт все зауваження повинні бути виправлені.

```
src/main.c:40:8: warning: implicit conversion loses integer precision:
'long' to 'unsigned int' [-Wshorten-64-to-32]
        srand((long)time(0));
        ~~~~~ ^~~~~~~~~~~~~
1 warning generated.
```

- неуспішна компіляція. Результуючий файл не створився. В результаті компіляції вказується в якому файлі та якій строчці є помилки.

```
src/main.c:10:18: error: use of undeclared identifier 'RAD'
                        result = PI * RAD IUS * RADIUS * HEIGHT;
                                      ^
1 error generated.
```

2.11.2 Запуск програми.

Для запуска програми ми маємо два шляхи:

- запуск напряму з поточного каталога кореня проекту: `./dist/main.bin`
- перехід до каталога dist та запуск виконуваючого файлу, вернутися на попередній каталог:

```
cd dist
./main.bin
cd ..
```

Так як наша програма нічого не виводить до екрану - проміжні результати отримаємо в розділі відлагодження.

2.11.3 Автоматизація дій. Утиліта make

Незважаючи на те, що ми навчилися компілювати та запускати програму, з часом стає питання автоматизації дії “компіляція-запуск”. Звичайним ctrl+c - ctrl+v на поточний час можна обійтись, але згодом проект стане більшим, кількість файлів для компіляції стане більшим та ін. І цього механізму перестане бути достатньо.

Наступним кроком є створення shell скриптів для кожної дії, але, як показує практика, вони стають об’ємними, незрозумілими та не достатньо ефективними.

На допомогу приходить утиліта make, що використовується у більшості Linux-додатків, та навіть при компіляції самого Limnux. Його інсталяція:

```
sudo apt install make
```

Утиліта make - це програмний інструмент для управління та підтримки комп’ютерних програм, що складається з багатьох компонентних файлів. Утиліта make автоматично визначає, які фрагменти великої програми потрібно перекомпілювати, і видає команди для їх перекомпіляції.

- make читає його інструкцію з Makefile (називається файлом дескриптора) за замовчуванням.
- Makefile встановлює набір правил, щоб визначити, які частини програми потрібно перекомпілювати, і видає команду для їх перекомпіляції.
- Makefile - це спосіб автоматизації процедури створення програмного забезпечення та інших складних завдань із залежностями.
- Makefile містить: правила залежності, макроси та суфіксні (або неявні) правила.

2.11.3.1 Правила залежності Правило складається з трьох частин, однієї або декількох цілей, нульової або більше залежностей та нульових чи більше команд у формі:

```
мета: залежності
<tab> команди для досягнення цілі
```

- `<tab>` символ НЕ МОЖЕ бути замінено пробілами.

- “ціль” (target), як правило, є ім’я файлу (наприклад, виконуваних файлів або об’єктних файлів). Це також може бути назва дії (наприклад, очистити)
- “залежності” - це файли, які використовуються як вхід для створення цілі.
- Кожна команда в правилі інтерпретується оболонкою, яку потрібно виконати.
- За замовчуванням make використовує /bin/sh оболонку.
- Виконання `make <target>` призведе до:
  1. Переконається, що всі залежності актуальні
  2. Якщо ціль не актуальна (out-of-date) за будь-яку залежність, відтворює її за допомогою вказаних команд.
- За замовчуванням введення “make” виконує першу ціль у Makefile.

2.11.3.2 Макроси

- Загальна інформація:
  - Використовуючи макроси, ми можемо уникнути повторення текстів, а makefile легко змінювати.
  - Визначення макросів мають вигляд `NAME = text string`, наприклад `CC=gcc` - визначаємо макрос CC, що буде вказувати на компілятор gcc.
  - На макроси посилаються, розміщуючи ім’я в круглих дужках або фігурних дужках та передуючи цьому знаку :`(CC) main.c -o prog.bin‘.
- Внутрішні макроси:
  - Внутрішні макроси заздалегідь визначені в утиліті make
  - `make -p` відображає список усіх макросів, суфіксних правил та цілей, що діють для поточної збірки.
- Спеціальні макроси:
  - Макрос `$@` відображає ім’я поточної цілі. Таким чином, наступні дві дії мають однаковий результат:

```
prog1 : $(objs)
    $(CXX) -o $@ $(objs)
    $(CXX) -o prog1 $(objs)
```

- Макроси командного рядка. Їх можна визначити в командному рядку, наприклад: `make DEBUG_FLAG=-g`, де DEBUG_FLAG - макрос, який можна використовувати в Makefile

2.11.3.3 Результуючий файл

Як працює make:

- Утиліта make порівнює час модифікації цільового файлу з часом модифікації файлів залежності. Будь-який файл залежності, який має останній час модифікації, ніж його цільовий файл, примушує відтворити цільовий файл.
- За замовчуванням перший цільовий файл - це той, що будується. Інші цілі перевіряються лише у тому випадку, якщо вони є залежностями для першої цілі.

- За винятком першої цілі, порядок цілей не має значення. Утиліта make будуватиме їх у необхідному порядку.

Таким чином, результуючий `Makefile` файл для сборці проекту має наступний вигляд:

```
targets = main.bin

CC = gcc

C_OPTS = -std=gnu11 -g \
    -Wall -Wextra -Werror -Wformat-security -Wfloat-equal -Wshadow \
    -Wconversion -Wlogical-not-parentheses -Wnull-dereference


all: clean prep compile run


clean:
    rm -rf dist

prep:
    mkdir dist

compile: main.bin


main.bin: src/main.c
    $(CC) $(C_OPTS) $< -o ./dist/$@

run: clean prep compile
    ./dist/main.bin
```

Він має наступні дії (цілі, targets):

- all - виконує весь реалізований життєвий цикл розробки продукту: підготовка результуючої директорії, компіляція, запуск
- clean - очистка проекту від тимчасових даних попередніх компіляцій
- prep - підготовка (створення) результуючої директорії
- compile - компіляція
- run - запуск скомпільованого файлу. При цьому, якщо файл буде відсутній, він буде створений викликом інших цілей: clean prep compile

Тобто:

- щоб скопилювати наш файл ми виконуємо команду `make compile` або `make clean prep compile`
- щоб виконати весь життєвий цикл розробки проекту , ми виконуємо `make all` або навіть `make` , т.я. ціль *all* є першою (а значить, за умовчуванням).

2.11.4 Робота з git

Виконав основу функціональну частину та переконавшись що вона працює, давайте відправимо наші зміни/функціонал в віддалений репозиторій, щоб викладач міг би їх передивитись:

```
$ cd ~/dev/project_linear/
$ git status
```

```
... Тут ми бачимо, що жоден файл не проіндексован ....
$ git add Makefile
$ git add src/main.c
$ git status
...
new file:   Makefile
new file:   src/main.c
$ git commit -m "Lab03 initial implementation"
$ git status
... Тут ми бачимо, що жоден файл вже не проіндексован  ...
... (усі файли закомічені ) ....
$ git push origin master
```

2.12 Питання для самоконтролю

1. Які символи входять до алфавіту мови програмування С?
2. Що таке лексема? Які бувають лексеми?
3. Що таке ідентифікатор? Які правила створення ідентифікаторів?
4. Що таке ключові слова? Які бувають ключові слова?
5. Які основні типи даних в мові програмування С?
6. Яке призначення цілого типу int і його специфікаторів?
7. Які типи даних у мові С для зберігання чисел із плаваючою крапкою?
8. Чим відрізняються змінні від іменованих констант?
9. Як можна оголосити константи у мові С?
10. Навіщо вказувати типи даних для констант та змінних?
11. Що таке тернарний оператор, як він виконується?
12. Які існують форми запису операцій інкремент і декремент? В чому їх відмінності?
13. Які порозрядні операції є у мові С, які правила їх виконання?
14. Що таке сполучені операції присвоювання, які бувають?
15. Яке правило перетворення типів без явного їх перетворення?
16. Які правила явного перетворення типів даних?
17. Яку назву має точка входу у програму?
18. Яку структуру має найпростіша програма?
19. Скільки точок входу може бути у програму?
20. Як оформляються коментарі у програмі?

3 Вступ до схем алгоритмів

Схема – графічне представлення визначення, аналізу або методу вирішення завдання, в якій використовуються символи (інакше, блоки) для відображення даних, потоку, устаткування і т.і.

Блок-схема – розповсюджений тип схем (графічних моделей), що описують алгоритми або процеси, у яких окремі кроки зображуються у вигляді блоків (символів) різної форми, з'єднаних між собою лініями, що вказують напрямок послідовності.

Блок-схеми алгоритмів при бажанні можна розглядати як графічну альтернативу псевдокоду. Одна з відмінностей полягає в тому, що для псевдокоду стандартів немає, а для блок-схем є.

Чинний міжнародний стандарт на блок-схеми (Вітчизняний стандарт ГОСТ 19.701-90) є точною копією міжнародного.

Існують і інші засоби для запису алгоритмів. У деяких англомовних підручниках поряд з ісевдокодом і блок-схемами використовуються діаграми Насс-Шнейдермана.

Створення блок-схем пов'язане з ім'ям видатного вченого Джона фон Неймана. Блок-схеми виявилися ефективним соціальним винаходом, які завоювали величезну популярність, вийшли далеко за межі інформатики і проникли майже в усі галузі знання.

3.1 Переваги та недоліки блок-схем

- блок-схеми алгоритмів - це єдина алгоритмічна мова, що зуміла подолати прірву між інформатикою та іншими дисциплінами і знайти застосування в багатьох (майже у всіх) областях людського знання.
- хоча при навчанні програмуванню псевдокод нерідко розглядається як основний засіб, проте, популярність блок-схем у багато разів перевищує популярність псевдокоду, якщо врахувати не тільки програмування і математику, а й інші галузі знання.
- візуальний синтаксис блок-схем алгоритмів, описаний в діючих стандартах, має серйозні недоліки. Він не має наукового (математичного і когнітивно-ергономічного) обгрунтування. Ця обставина є серйозним недоліком блок-схем, який різко обмежує можливості людей і створює нездоланний бар'єр при роботі зі складними алгоритмами.
- блок-схеми здатні проникнути в багато області знання, проте вони придатні для роботи тільки з простими алгоритмами. Блок-схеми не дозволяють зрозуміло висловити зміст складних алгоритмів. З ростом складності блок-схеми стрімко втрачають наочність і стають практично нежиттєздатними, особливо, для непрофесіоналів. Для більш складних алгоритмів існують інші засови відобреження, наприклад, Діаграми послідовностей, UML діаграми та ін..

- блок-схеми алгоритмів та інші традиційні засоби не дозволяють подолати нинішню масову алгоритмічну неграмотність.

3.2 Причини масової алгоритмічної неграмотності

У нашому житті алгоритми відіграють надзвичайно важливу роль. Ми стикаємося з ними на кожному кроці. Вони оточують нас всюди - від космічних ракет до бухгалтерських розрахунків. Вони примудряються жити в кожній живій клітині - від холерного вібріона до незабудки і слона.

Але ось біда, для більшості людей алгоритми - це загадкові невидимки. Лише мало хто знає, що успіхи цивілізації у величезній мірі залежать від алгоритмів. Якщо якийсь злий чарівник знищить алгоритми, світова економіка звалиться. І все живе на планеті кане в небуття.

На жаль, фахівці-непрограмісти (хіміки, біологи, технологи, економісти та ін.) майже нічого не знають про алгоритми. Вони не вміють розчленовувати свої знання і виділяти серед них алгоритмічну (процедурну) частину. Вони не в змозі висловити свої знання на папері у формі алгоритмів.

Нинішні алгоритми, використовувані в усьому світі, мають серйозний дефект. Вони надзвичайно важкі для розуміння. Існуюча практика вивчення, розробки та експлуатації алгоритмів є незадовільною. Вона вимагає невиправдано великих витрат праці і часу.

Алгоритмічна необізнаність фахівців-непрограмістів пояснюється тим, що вивчення алгоритмів є занадто складним і навіть непосильною справою. Тому робота з алгоритмами для переважної більшості професіоналів виявляється неможливою.

3.3 Алгоритмизация чи програмування?

- алгоритмізація і програмування - істотно різні речі.
- масове навчання програмуванню неможливо і не потрібно з двох причин. По-перше, воно неймовірно важко. По-друге, воно дає знання, які більшості просто не потрібні.
- масове навчання алгоритмізації, навпаки, корисно і необхідно.
- число людей, яким необхідно знати алгоритми, у багато разів перевищує число людей, яким треба знати програмування.

Уміння формалізувати власні процедурні знання фахівців-непрограмістів, а також вміння розробляти і зображати алгоритми у своїй предметній області має стати частиною їх професійної культури.

3.4 Правила застосування символів

Правила виконання блок-схем регламентуються ДСТУ ISO 5807:2016 “Обробляння інформації. Символи та угоди щодо документації стосовно даних,

програм та системних блок-схем, схем мережевих програм та схем системних ресурсів". Даний ДСТУ практично повністю відповідає міжнародному стандарту ISO 5807:1985.

Згідно із прийнятими стандартами, усі схеми алгоритмів відображаються за принципом "зверху-вниз", тобто побудована схема повинна бути витягнута максимально вертикально. При розгалуженні рекомендується розширення схем виконувати по горизонталі в праву сторону. Символи в схемі повинні бути розташовані рівномірно. Варто дотримуватися розумної довжини з'єднань і мінімального числа довгих ліній.

При зображенні елементів рекомендується дотримуватися розмірів, визначених двома значеннями a і b (ширина і висота символу). Значення a (ширина) вибирається з ряду 15, 20, 25, … мм; b (висота) розраховується для кожного компоненту окремо та буде наведено нижче. Символи повинні бути, по можливості, одного розміру. Визначення розмірів несе рекомендаційний характер, проте, при дотриманні обраних розмірів блок-схеми мають більш акуратний вигляд.

Текст всередині елементів схеми має бути Times New Roman 12 пт, коментарі та нумерація блоків – Times New Roman 10 пт.

У середині символу варто поміщати мінімальну кількість тексту, яка необхідна для розуміння функції даного символу. Якщо обсяг тексту перевищує розміри символа, варто використовувати символ коментарю. Розповсюдженою та помилковою практикою є спроба використання блок-схем для ілюстрації алгоритму на низькому рівні (на рівні коду) – тобто, спроба вписувати в блоки схеми фрагменти коду на будь-якій штучній мові. Такий підхід можна застосовувати тільки до програм, організованих згідно зі структурним підходом, і не можна, наприклад, до алгоритму, який реалізує взаємодію абстракцій при об'єктно-орієнтованому підході.

У схемах може використовуватися ідентифікатор символів, який використовується для посилань на них у документах при їх словесному описі поза межами рисунку. Ідентифікатор символу повинний розташовуватися ліворуч над символом и складатися з цифр або букв та цифр.

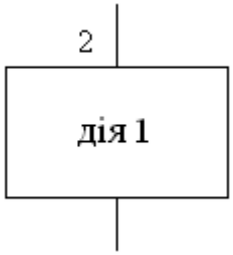

Рисунок 1 – Ідентифікація символів

При складанні схем алгоритмів слід дотримуватися правила: нічого не виходить нізвідки і не йде в нікуди. Схема алгоритму має бути замкненою, "висячих" стрілок, а також стрілок, які з'явилися з нізвідки, не може бути.

3.5 Правила виконання з’єднань

Потоки даних у схемах показуються лініями. Напрямок потоку зліва напра-во і зверху униз вважається стандартним; в таких випадках на лініях стрілки ставить необов’язково. У випадках, коли необхідно внести ясність у схему (наприклад, при з’єднаннях), на лініях використовуються стрілки. Якщо потік має напрямок, відмінний від стандартного, стрілки повинні вказувати цей напрямок.

У схемах варто уникати перетинання ліній. Дві або більше вхідних ліній можуть поєднуватися в одну вихідну лінію, в цьому випадку місце об’єднання повинне бути зміщеним.

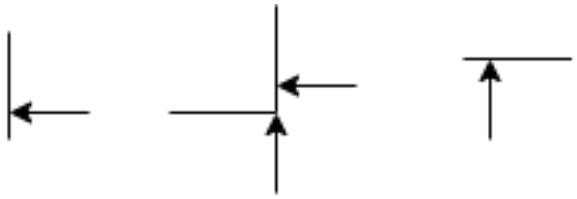

Рисунок 2 – Приклад з’єднань потоків

Лінії в схемах повинні підходити до символу або з лівого боку, або зверху, а виходити або правого боку, або знизу. Лінії мають бути спрямовані до центра символу.

При необхідності лінії в схемах варто розривати для запобігання зайвим перетинанням або занадто довгим лініям, а також у випадках, коли схема розміщується на декількох сторінках. В таких випадках використовуються з’єднувачі. З’єднувач на початку розриву називається зовнішнім з’єднувачем, а з’єднувач наприкінці розриву – внутрішнім з’єднувачем. З‘єднувач є парним символом. Внутрішній та його зовнішній з’єднувачі повинні мати однакові ідентифікатори. У разі, коли зовнішній з’єднувач розташовується на іншому листі, посилання на лист рекомендується наводити разом із символом коментаря для їхніх з’єднувачів.

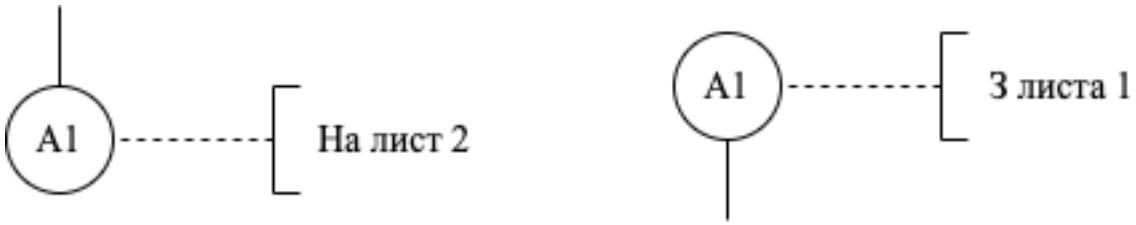

Рисунок 3 – Приклад конекторів

3.6 Використання символів

3.6.1 Термінатори

Термінатори використовуються для відображення початку або кінця програми або підпрограми (функції). Висота приблизно дорівнює 0,4 ширини.

Термінатор початку має тільки один вихід знизу та має текст “Початок”. Термінатор кінця має тільки 1 вхід зверху та має текст “Кінець”.

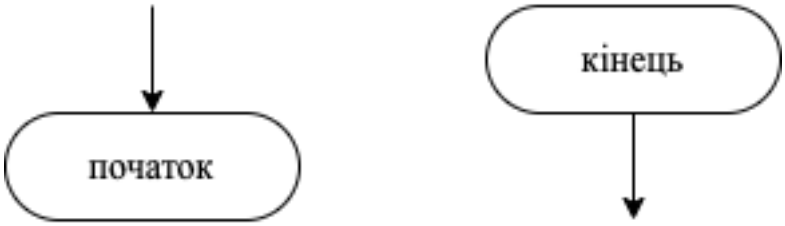

Рисунок 4 – Термінатори

3.6.2 Коментарі

Елемент використовується для детальнішої інформації про кроки, процесу або групи процесів. Опис поміщається з боку квадратної дужки і охоплюється нею по всій висоті. Пунктирна лінія йде до описуваного елементу, або групи елементів (при цьому група виділяється замкнутою пунктирною лінією). Висота блоку дорівнює приблизно 0,75 ширини. Також символ коментаря слід використовувати в тих випадках, коли обсяг тексту в будь-якому іншому символі (наприклад, символ процесу, символ даних та ін.) перевищує його обсяг.

Рисунок 5 – Блок коментарів

3.6.3 Блок дії або процесу

Виконання однієї або декількох операцій, обробка даних будь-якого виду (зміна значення даних, форми подання, розташування). Висота блоку дорівнює приблизно 0,75 ширини. Усередині фігури записують безпосередньо самі операції, наприклад, операцію присвоювання. Формули виносяться в коментарі. Блок дії не містить у собі оголошення, ініціалізацію змінних! Даний блок має один вхід зверху й один вихід знизу.

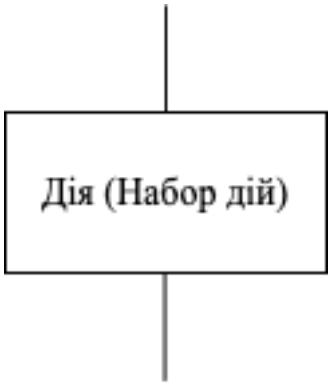

Рисунок 6 – Блок дії

3.6.4 Символ введення-виведення

Перетворення даних у форму, придатну для обробки (введення) або відображення результатів обробки (виведення). Цей символ не визначає носія даних (для вказівки типу носія даних використовуються специфічні символи), тобто не має значення, чи ми працюємо з консоллю, файлом чи іншими системами введення-виведення. Висота блоку дорівнює приблизно 0,75 ширини.

Рисунок 7 – Блок введення/виведення

3.6.5 Символ з умовою (розгалуження)

Символ відображає оброку умови, рішення або функцію перемикального типу з одним входом і двома або більше альтернативними виходами, з яких тільки один може бути обраний після обчислення умови, яка визначена всередині цього символу. Висота символу дорівнює 0.75 ширини. Вхід в символ позначається лінією, що входить зазвичай у верхню його вершину. Даний блок має завжди два виходи. Зазвичай кожен вихід позначається лінією, що виходить з решти вершин (бічних і нижньої).

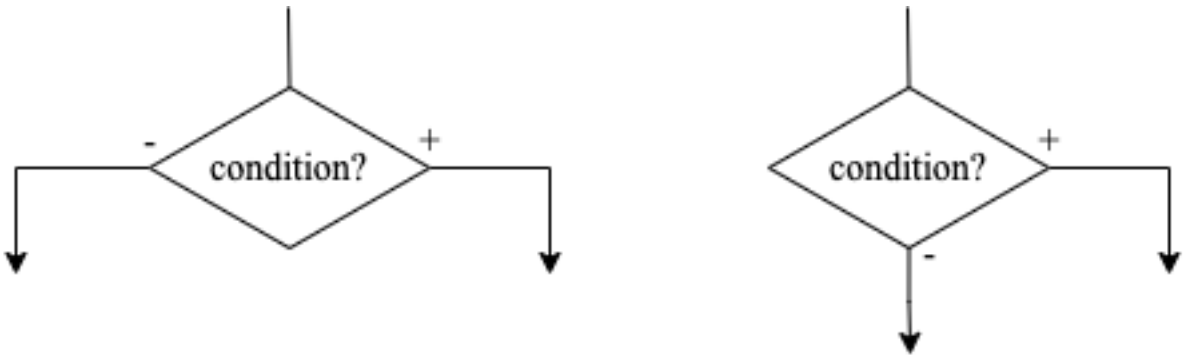

Рисунок 8 – Блок розгалуження

Якщо виходів більше двох (наприклад, оператор вибору switch-case), то їх слід показувати однією лінією, що виходить з вершини (частіше нижньої) символу, яка потім розгалужується. Кожен вихід із символу повинен супроводжуватися відповідними значеннями умов (результатами обчислень),

щоб показати логічний шлях, що він представляє, для того, щоб ці умови і відповідні посилання були ідентифіковані.

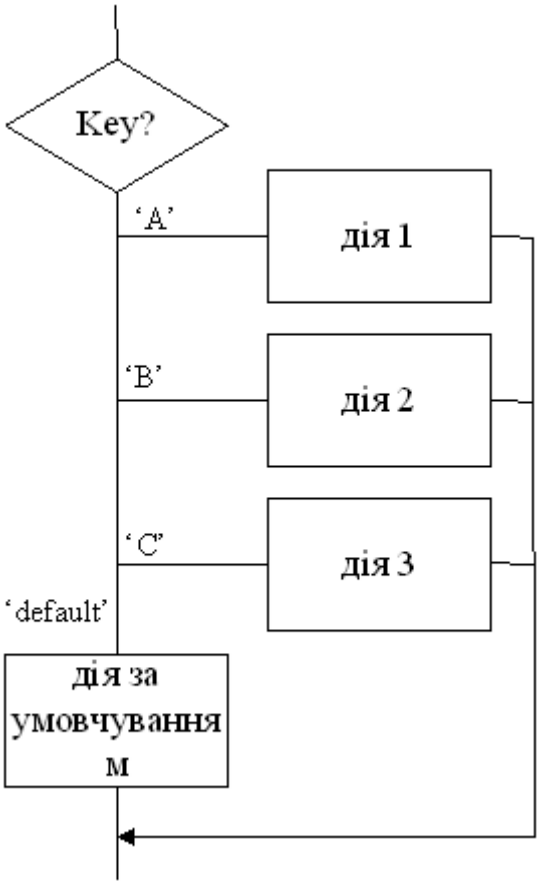

Рисунок 9 – Оператор вибору switch-case

3.6.6 Межі циклу

Елемент складається з двох частин – відповідно, початок і кінець циклу; операції, що виконуються всередині циклу, розміщуються між ними. Умова циклу та збільшення/зменшення змінної, по якій організовано цикл, записуються всередині символу початку або кінця циклу в залежності від типу організації циклу Для оператора циклу з передумовою умова записується в середині символу початку, а змінна, по якій організовано цикл, – в середині символу кінця. Для циклу с післяумовою навпаки – умова записується в символі кінця.

Часто для зображення циклу на блок-схемі замість цього символу використовують символ рішення, вказуючи в ньому умову, а одну з ліній виходу замикають вище в блок-схемі (перед операціями циклу).

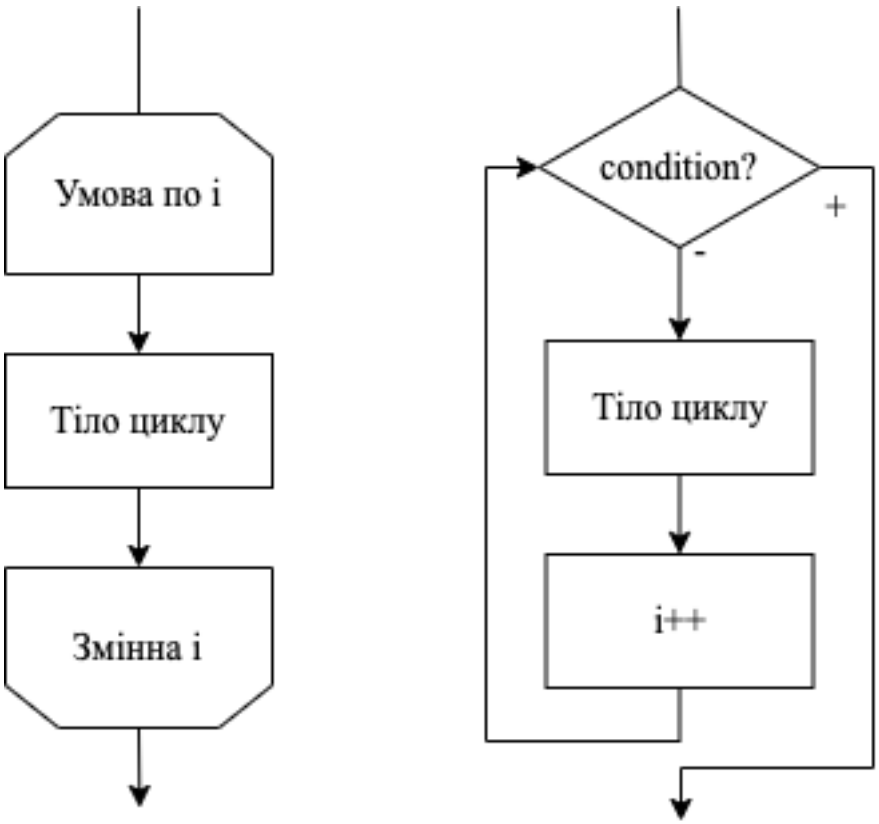

Рисунок 10 – Межі циклу

3.6.7 Символ модифікації (циклу for)

Символ модифікації використовують для відображення заголовка циклу з параметром. У ньому через крапку з комою вказуються ім’я змінної (параметра) з початковим значенням, граничне значення параметра (або умова виконання циклу), крок зміни параметра. На рисунку подано приклад символу модифікації для відображення цикла з параметром і, що змінюється від 0 до 10 з кроком 2. Якщо шаг дорівнює 1, то його на схемі можна не відображати.

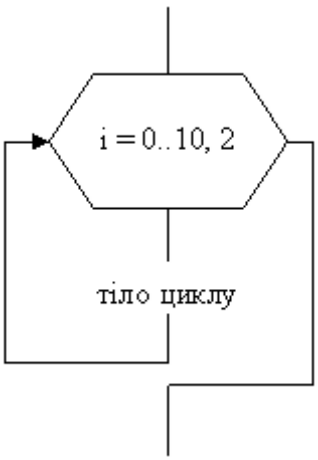

Рисунок 11 – Приклади циклу for

3.6.8 Символ зі смугою (функція)

В схемах може використовуватися докладне подання, що для процесу позначається за допомогою символу зі смугою. Символ зі смугою вказує, що в цьому ж комплекті документації але в іншому місці надається більш докладне подання даного процесу. Символ зі смугою – це символ, усередині якого у верхній частині проведена горизонтальна лінія. Між цією лінією і верхньою лінією символу поміщений ідентифікатор, що вказує на докладне подання даного символу.

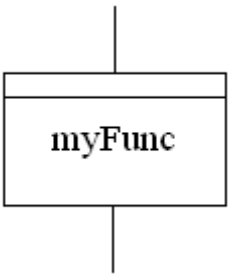

Рисунок 12 – Символ виклику функції

Докладне подання починається термінатором початку, в середині якого замість слова “Початок” повинне міститися назва функції. Закінчується докладне подання термінатором кінця.

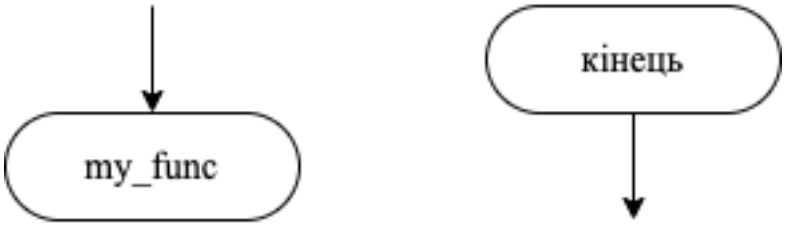

Рисунок 13 – Символи початку та кінця функції

При використанні раніше створених і окремо описаних алгоритмів або програм, наприклад, процедури стандартної бібліотеки, використовується символ зумовленного процесу. Усередині символу записується назва процесу і передані в нього дані. Якщо переданих даних багато, то вони виносяться в коментарі. Якщо функція повертає результат, який використовується далі, то рекомендується вказати, що результат заноситься в задану змінну.

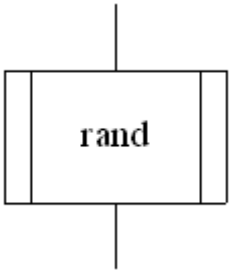

Рисунок 14 – Символ виклику раніше створеної функції

3.7 Приклади розробки схем алгоритмів

3.7.1 Обчислення площі прямокутника

Завдання: по заданих сторонах прямокутника визначити його площу.

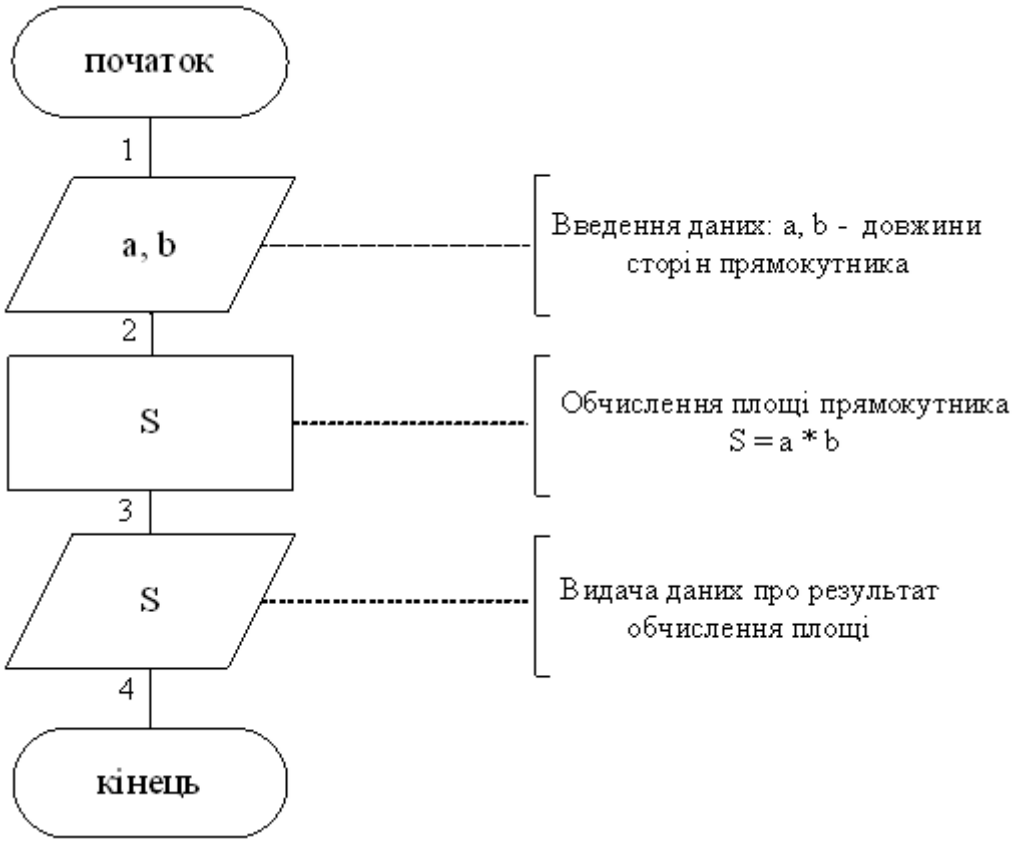

Рисунок 15 – Схема алгоритму обчислення площи прямокутника

Опис алгоритму: вводяться значення сторін прямокутника (блок1), обчислюється його площа (блок 2), видається результат (блок 3).

3.7.2 Визначення мінімального числа

Завдання: серед трьох заданих чисел знайти найменше серед них.

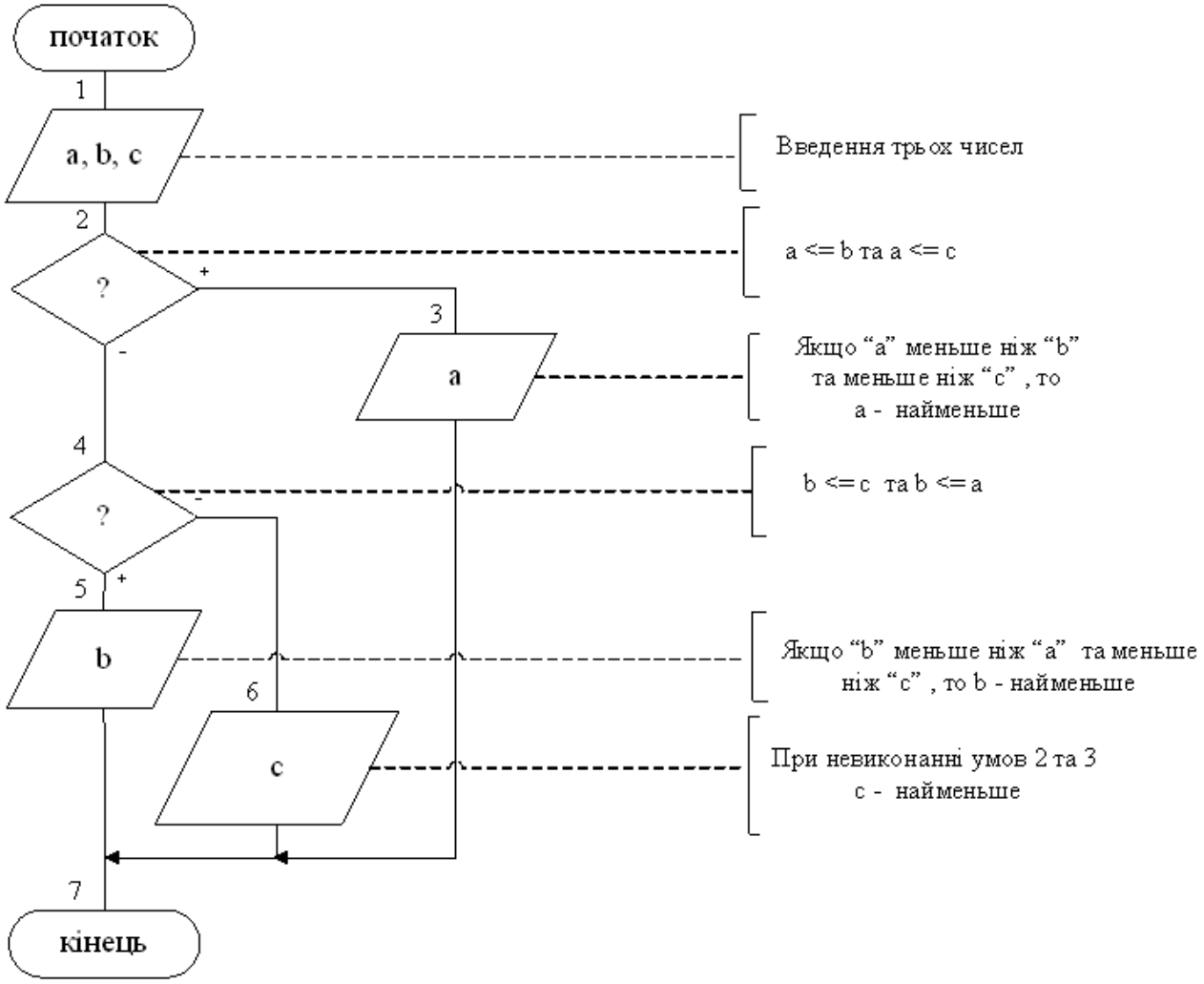

Рисунок 16 – Схема алгоритму обчислення мінімального числа

3.7.3 Абстрактна міні-гра

Завдання: є уявний персонаж, якого треба рухати шляхом натискання на кнопки W, A, S, D (вперед/вправо/назад/вліво). Дія буде відбуватися "абстрактно" (без конкретної реалізації), після чого буде видаватися запит на одержання наступної "команди". При натисканні на кнопку Q – програма повинна видати повідомлення про закінчення і зупинитися. При натисканні на клавішу, яка не є W, A, S, D або Q – програма нічого не виконує і знову видає запит на введення наступної дії.

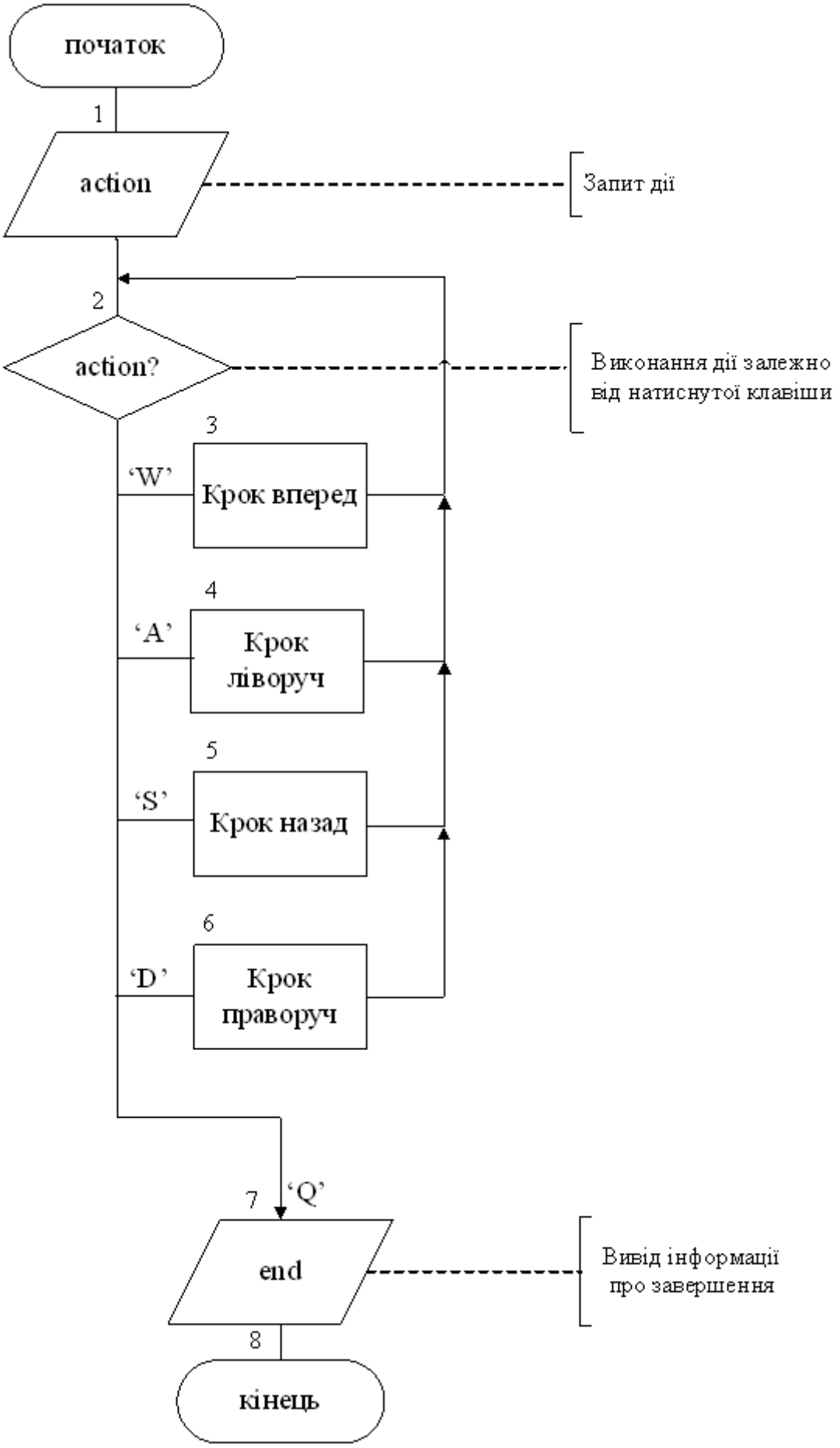

Рисунок 17 – Схема алгоритму абстрактної ігри

Опис алгоритму: В блоці 1 видається запит на введення команди. Команда аналізується в блоці 2 і в залежності від натиснутої клавіші виконуються оператори в блоках 3, 4, 5 або 6. Після натискання на клавішу Q, програма закінчує свою роботу.

3.7.4 Обчислення факторіала

Завдання: обчислити факторіал введеного числа.

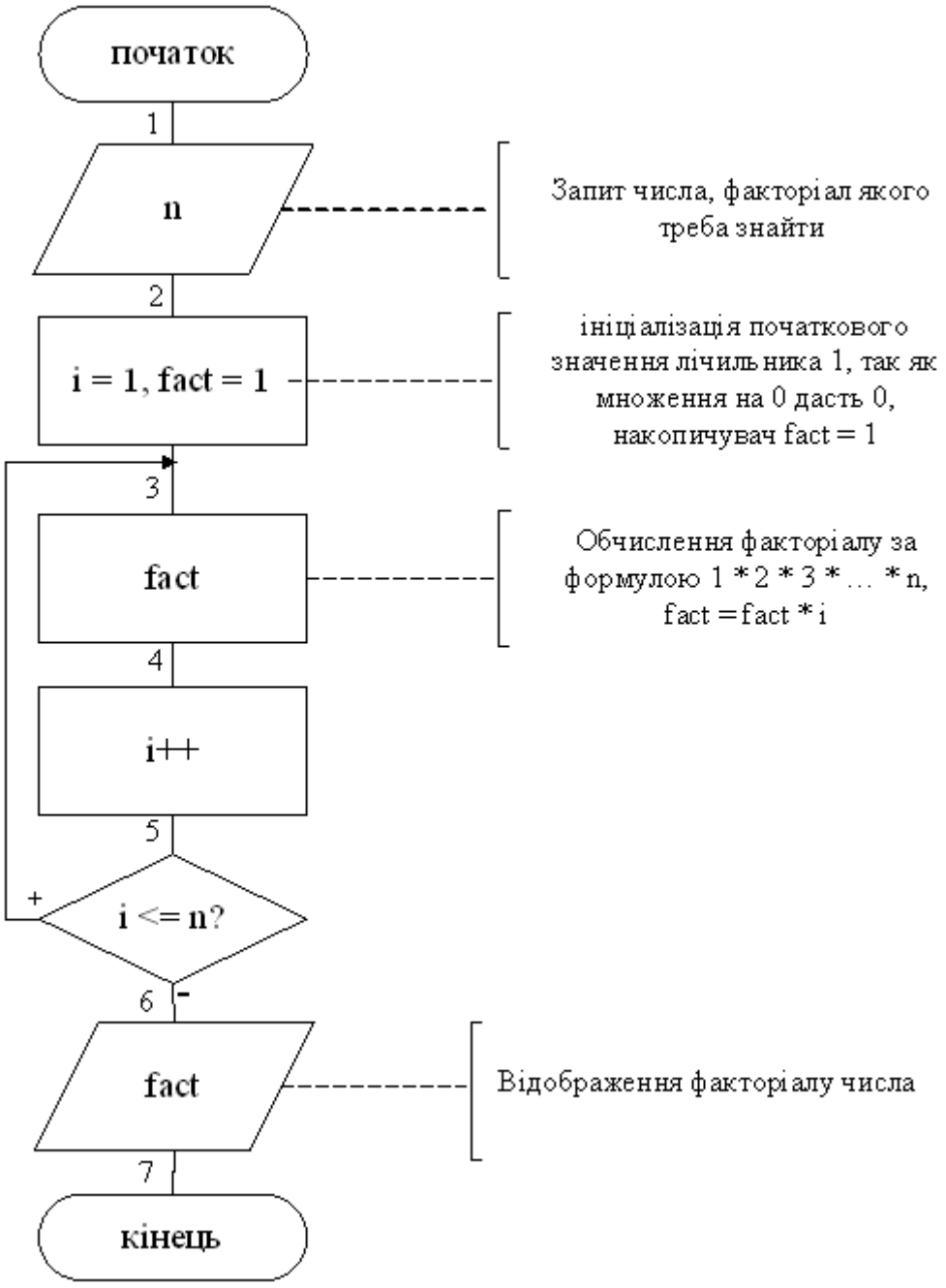

Рисунок 18 – Схема алгоритму функції main() із використанням циклу do-while

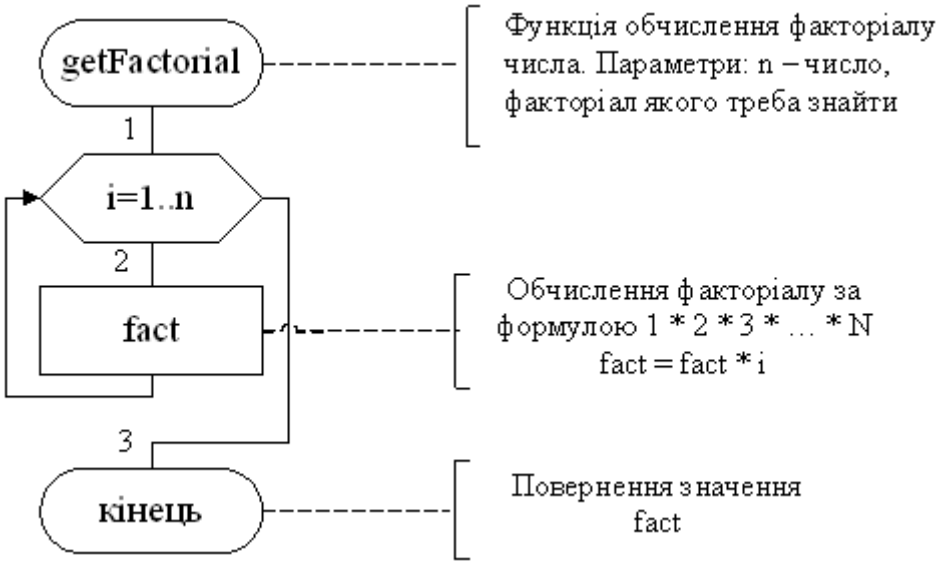

Рисунок 19 – Схема алгоритму функції getFactorial() із використанням циклу for

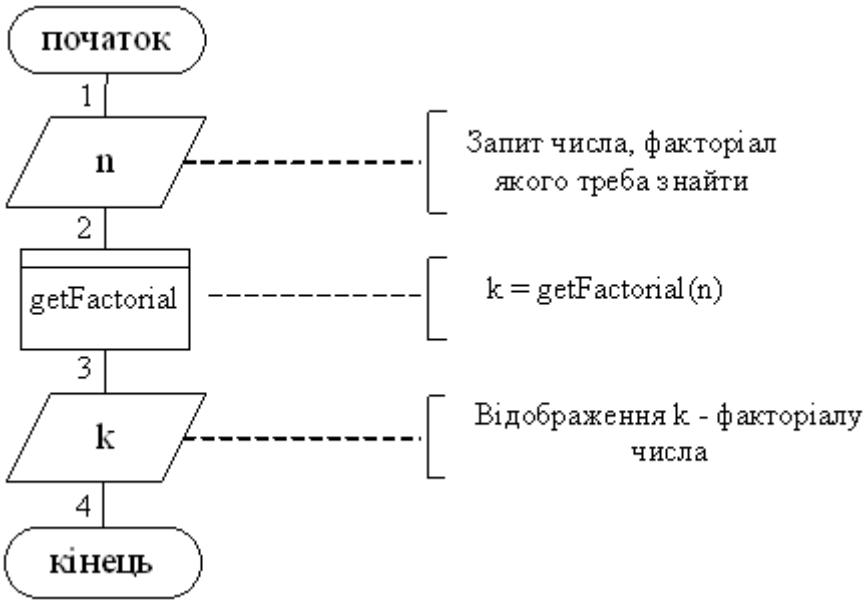

Рисунок 20 – Схема алгоритму функції main(), яка обчислює факторіал числа шляхом виклику функції getFactorial()

3.7.5 Обчислення суми чисел

Завдання: для заданого n визначити суму чисел з інтервалу від 1 до n.

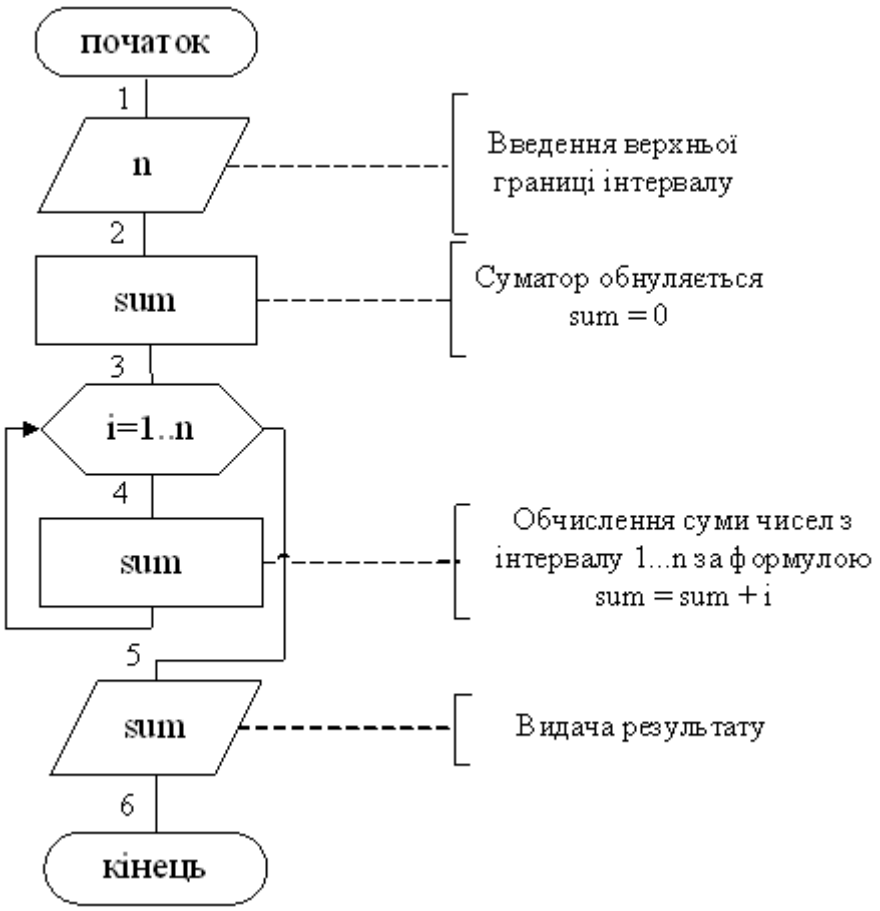

Рисунок 21 – Схема алгоритму обчислення суми чисел

Опис алгоритму: вводиться значення верхньої межі інтервалу, суму чисел якого треба обчислити (блок 1). В блоці 2 суматору sum привласнюється початкове значення 0. В циклі (блоки 3,4) до суматора додається число i, яке приймає значення 1, 2, …, n. Обчислене значення суми видається в блоці 5.