МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

ПРОГРАМУВАННЯ ТА АЛГОРИТМІЧНІ МОВИ 1
АЛГОРИТМІЗАЦІЯ ТА ОСНОВИ ПРОГРАМУВАННЯ

Конспект лекцій

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів,
які навчаються за спеціальністю 124 «Системний аналіз»,
освітньо-професіними програмами «Системний аналіз та управління»,
«Системний аналіз фінансового ринку»

Київ
КПІ ім. Ігоря Сікорського
2020

ПРОГРАМУВАННЯ ТА АЛГОРИТМІЧНІ МОВИ 1. АЛГОРИТМІЗАЦІЯ ТА ОСНОВИ ПРОГРАМУВАННЯ: Конспект лекцій [Електронний ресурс]: навч. посіб. для студ. спеціальності 124 «Системний аналіз», освітньо-професійні програми «Системний аналіз та управління», «Системний аналіз фінансового ринку» / КПІ ім. Ігоря Сікорського; уклад.: І.В. Назарчук. – Електронні текстові дані (1 файл: 1,1 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2020. – 140 с.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 6 від 31.01.2020 р.) за поданням Вченої ради Інституту прикладного системного аналізу (протокол №1 від 27.01.2020 р.)

Електронне мережне навчальне видання

ПРОГРАМУВАННЯ ТА АЛГОРИТМІЧНІ МОВИ 1 АЛГОРИТМІЗАЦІЯ ТА ОСНОВИ ПРОГРАМУВАННЯ
Конспект лекцій

Укладач: *Назарчук Ірина Василівна*

Відповідальний редактор *Тимощук О.Л.*, к.т.н., доцент

Рецензент: Гагарін О.О., к.т.н., доцент, доцент кафедри АПЕПС ТЕФ НТУУ «КПІ ім.І.Сікорського»

У конспекті лекцій викладені основні поняття програмування та алгоритмічних мов, а саме: основні конструкції мови програмування на прикладах С та С++, основні структури даних, засоби стандартних бібліотек, базові принципи структурного програмування. Надані приклади використання конструкцій мови із залученням стандартних алгоритмів та базових знань з початкових курсів вищої математики.

Конспект лекцій складено відповідно до навчального плану дисципліни «Програмування та алгоритмічні мови», кредитного модуля «Алгоритмізація та основи програмування» і призначений для студентів спеціальності 124 «Системний аналіз». Викладення матеріалу передбачає одночасне вивчення студентами дисциплін «Дискретна математика», «Алгоритми і структури даних», «Математична логіка і теорія алгоритмів».

Зміст

Вступ

Дисципліна «Програмування та алгоритмічні мови» належить до циклу загальної підготовки, має статус обов'язкової і є однією з основних у підготовці студентів першого бакалаврського рівня вищої освіти спеціальності 124 «Системний аналіз» і є базовою для таких дисциплін як «Об'єктно-орієнтоване програмування», «Бази даних», «Архітектура обчислювальних систем», «Розробка і тестування програм», а також для усіх спеціальних курсів, що потребують комп'ютерного моделювання.

У даному конспекті лекцій подано теоретичний матеріал за розділами: «Загальна характеристика програмного забезпечення», «Базові елементи та конструкції мови», «Вказівники», «Структурований тип масив», «Символьні рядки», «Функції та механізм передачі параметрів», «Структури та об'єднання», відповідно до навчальної програми кредитного модуля «Програмування та алгоритмічні мови 1 Алгоритмізація та основи програмування».

Викладення матеріалу згруповане за темами лекцій. Кожне нове поняття, що уводиться в лекції, проілюстроване фрагментами програм мовою С, складні для розуміння теми – прикладами завершених програм з коментуванням коду і отриманих результатів.

Матеріал, який передбачає використання довідникової інформації, доповнений відповідними таблицями, великі таблиці винесені у додатки.

Для більшої наочності окремі теми містять графічну інформацію.

Використаний математичний апарат базований на шкільному курсі математики і початкових темах з математичного аналізу та лінійної алгебри. Кращому розумінню матеріалу лекцій сприяє одночасне вивчення студентами дисциплін «Дискретна математика» , «Алгоритми і структури даних», «Математична логіка і теорія алгоритмів».

Розділ 1 Загальна характеристика програмного забезпечення

Лекція №1

Тема лекції: Обчислювальна система

План лекції

1. Обчислювальна система
2. Принципи роботи комп'ютера за фон Нейманом
3. Пам'ять комп’ютера
4. Системи команд
5. Характеристика програмного забезпечення (ПЗ)

Зміст лекції

1.1.Обчислювальна система

Обчислювальна система – програмно-апаратний комплекс, призначений для надання послуг користувачу.

Складається з апаратних засобів (*hardware*) та програмного забезпечення (*software*).

До *апаратного забезпечення* обчислювальних систем відносять пристрої та прибори, які утворюють апаратну конфігурацію. Це фактично базовий рівень, на якому можуть бути створені багато вищих рівнів розробленням і завантаженням відповідного програмного забезпечення.

Опис сукупності пристроїв та блоків таких систем і зв’язків між ними визначають *архітектуру* конкретного обчислювального пристрою. Поняття архітектури тісно пов’язане з принципами його роботи і визначає принципи дії, інформаційні зв’язки і взаємодію основних складових, а саме: процесора, внутрішньої і зовнішньої пам’яті та периферійних пристроїв. Уніфікація архітектури обчислювальних пристроїв забезпечує їх сумісність з погляду користувача.

1.2.Принципи роботи комп'ютера за фон Нейманом

Принципи, згідно з якими функціонує більшість сучасних комп’ютерів, опубліковано в 1946 році американським математиком Джоном фон Нейманом (1903–1057).

Основні принципи архітектури комп’ютерів фон Неймана є такі:

– принцип двійкового кодування полягає у тому, що усі дані подають у вигляді двійкових кодів;

– принцип програмного керування полягає у тому, що всі операції з опрацюванням даних здійснюють відповідно до програм і ці програми розташовують у пам’яті комп’ютера;

– принцип однорідності пам’яті полягає у тому, що всі дані, у тому числі й програми, зберігають в одному і тому самому пристрої пам'яті;

– принцип адресності полягає у такій організації пам’яті комп’ютера, за якої процесор може безпосередньо звернутись до даних, розташованих у будь-якій частині пам’яті, а кожна комірка пам’яті має унікальну назву – адресу;

Незважаючи на еволюцію обчислювальної техніки і незалежно від відмінностей стосовно способів фізичної реалізації, типовий персональний комп’ютер містить у своєму складі такі основні пристрої, наведені ще у класичній роботі фон Неймана, як: пристрій керування (ПК), арифметико-логічний пристрій (АЛП), пам’ять, система введення та система виведення даних. Схематично це можна уявити так, як показано на рисунку 1:

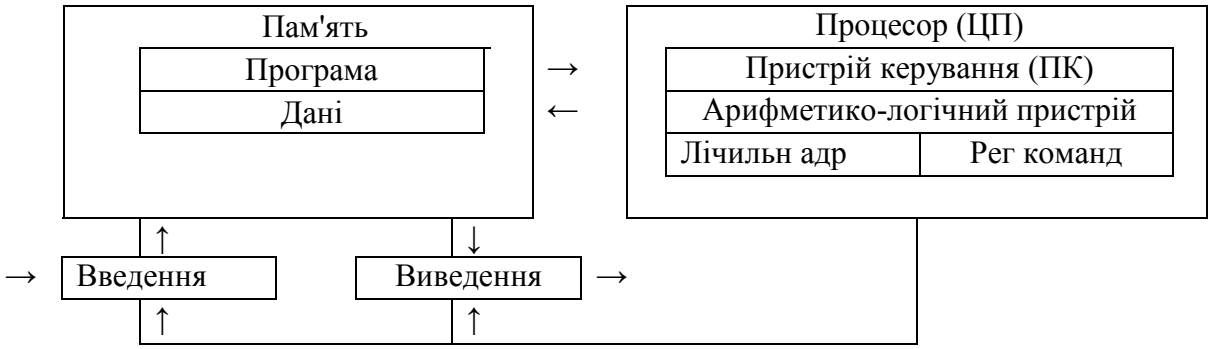


Рисунок 1. Принцип функціонування комп'ютера

Призначення цих пристроїв наступне.

1. За допомогою пристроїв введення дані і програми їх опрацювання потрапляють у пам’ять комп’ютера.

2. З пам’яті комп’ютера дані надсилають до процесора (англійською *central processing unit – CPU* – модуль центрального процесора), до складу якого і входять АЛУ та ПК.

3. Арифметично-логічний пристрій здійснює опрацювання даних.

4. Пристрій керування керує процесами опрацювання даних, їх збереженням і передаванням.

5. Пристрої виведення даних здійснюють подання результатів опрацювання даних у зручному для користувача вигляді.

1.3.Пам'ять комп’ютера

Пам'ять – це пристрій для збереження інформації з метою оперативного обміну нею із зовнішніми пристроями.

Розрізняють два типи пристроїв пам'яті.

Пристрої внутрішньої швидкої пам'яті – це оперативна пам’ять (ОП чи *RAM – read access memory*), регістри процесора та надшвидка кеш-пам’ять (*cash*), потрібні для зберігання опрацьовуваних даних. Вони є енергозалежними і при вимиканні живлення інформація в цих пристроях втрачається.

Пристрої пам'яті для довготривалого збереження інформації енергонезалежні, мають велику ємність, але не можуть забезпечувати швидкий доступ . Найпоширенішими сьогодні є жорсткі диски (вінчестери, HDD – hard disk driver), CD, DVD, USB-driver (“флеш-диски”), memory cards тощо;.

Регістри процесора – найбільш швидкісна, але найменша за обсягом пам'ять.

Основна пам'ять підрозділяється на постійну (*read only memory, ROM*) і оперативну (*random access memory, RAM*).

Оперативна пам'ять організована у вигляді множини комірок, кожна з яких містить деяку символьну або числову інформацію і має свою унікальну адресу, як показано на рисунку 2.

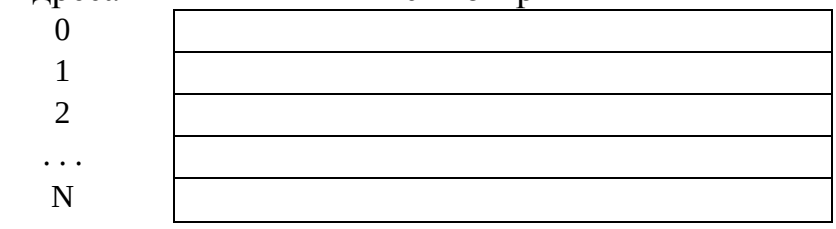


Рисунок 2. Адресація комірок пам'яті

Усі комірки RAM послідовно пронумеровані з нуля. Якщо *j*-та комірка містить число *m*, говорять, що «*комірка з адресою j має значення m*», або що її

«*вміст* дорівнює *m*». Кожна комірка неподільна і містить машинне слово, фізичний розмір якого залежить від розрядності комп’ютера.

Кеш-пам'ять розташована між регістрами і процесором і призначена для узгодження швидкостей процесора і основної пам'яті.

Зовнішня пам'ять передбачає наявність накопичувача, який представляє собою пристрій для уведення-виведення інформації. Накопичувачі бувають двох різновидів – для тривалого зберігання інформації та для взаємодії із зовнішнім середовищем (клавіатура, монітор, тощо).

1.4.Системи команд

Кожний процесор має набір вбудованих так званих машинних операцій. Це елементарні дії, виконувані апаратно, реалізовані у вигляді електричних схем. Їх кількість, перелік і структура визначені ще під час проектування комп’ютера.

Машинна команда – це наказ на виконання такої операції, *формат команди* – це правило її запису, а перелік усіх машинних операцій разом з форматами команд утворюють *систему команд процесора*.

Як правило, до системи команд входять арифметичні, пересилання та переходу. Комп’ютер працює по крокам – тактам. *Такт роботи процесора* – це виконання однієї машинної команди.

Якщо абстрагуватися від роботи зовнішніх пристроїв і вважати, що програма уже якимось чином занесена в комірки пам'яті, адреси комірок з даними відомі, то схема роботи процесора наступна.

Вмикання комп’ютера приводить до запису у лічильник адрес (ЛА) адреси першої команди, розташованої в ОЗП – це завжди фіксоване значення, наприклад, 0.

Після цього ЦП починає послідовно крок за кроком виконувати команди, поки не зустріне команду зупинки.

На кожному такті будуть виконані наступні дії:

1. до регістру команд (РК) заноситься вміст комірки, що знаходиться за адресою із ЛА;
2. значення ЛА збільшується на 1 і тепер вказує на наступну команду;
3. ПК розшифровує команду із РК і забезпечує її виконання;

Ця послідовність повторюється до команди зупинки ЦП.

1.5.Характеристика програмного забезпечення (ПЗ)

Програмне забезпечення (ПЗ) (software) – сукупність програм та службових даних, призначених для керування роботою комп'ютера. ПЗ комп'ютерів можна поділити на такі основні класи: системне ПЗ, прикладне ПЗ та системи програмування.

Системне ПЗ (*system software*) – програми та програмні комплекси спільні для використання усіма користувачами технічних засобів комп'ютера, застосовуються і для автоматизації розробки нових програм і для організації виконання існуючих.

До складу системного ПЗ входять:

1. Операційна система (ОС)
2. Система управління файлами (СУФ)
3. Операційне середовище
4. Утіліти
5. Системи програмування

Операційна система (ОС) – комплекс керуючих та обробляючих програм, який виконує функцію *інтерфейсу* між апаратною складовою та програмами користувача, забезпечує найбільш *ефективне* використання ресурсів обчислювальної системи й організацію *надійних* обчислень. Фактично це прошарок між апаратною складовою і рештою ПЗ (функції, класифікація).

СУФ призначена для *організації* зручного доступу до даних, організованих як файли. *Файл* – це іменований фрагмент фіксованого розміру на зовнішньому носії, що зберігає, як правило, семантично пов'язану між собою інформацію. *Файлова система* визначає засоби збереження інформації та правила доступу до неї, регламентує формат файлу.

Для роботи з файлами конкретної файлової системи під управлінням конкретної ОС розробляється відповідна СУФ. Завдяки СУФ замість доступу на низькому рівні з вказанням фізичних адрес комірок, використовують логічний доступ з вказуванням назви файлу і номера запису в ньому.

Операційне середовище – інтерфейс, необхідний програмам для звертання до ОС з метою отримання певного сервісу. Може включати декілька інтерфейсних оболонок.

Інтерфейсні оболонки призначені для *розширення* керуючих можливостей ОС, або *заміни* деяких з них для даної системи. Наприклад, графічний інтерфейс XWindow у системах Unix або інтерфейси для Windows, які замінюють Explorer і можуть нагадувати Unix, Mac, тощо .

Для моделювання в одній операційній системі іншої призначені так звані *емулятори*, наприклад в Linux можна запустити Windows за допомогою системи емуляції WMWARE, DOS Box для емуляції MS DOS у Windows, тощо.

Для запуску програм, створених під іншу операційну систему, для деяких ОС є можливість організувати *віртуальну машину.*

Утиліти – спеціальні невеликі системні програми, які виконують сервісні функції. Можуть бути організовані у вигляді інтегрованого середовища.

Прикладне ПЗ призначене для вирішення конкретних задач користувача. За формою це можуть бути окремі програми, пакети програм, бібліотеки, інтегровані середовища. Мають свою класифікацію за сферою використання: загального призначення, проблемно-орієнтовані, методо-орієнтовані, комунікативні.

Системи програмування (СП) – це системи розробки нових програм на конкретних мовах програмування. Сучасні інструментальні засоби розробки підтримують усі технологічні етапи розробки програми.

Мови програмування використовують для переведення алгоритмів у комп'ютерні програми для розробки системного та прикладного ПЗ. Мови програмування розподіляють на дві основні категорії – машинно-орієнтовані мови низького рівня і мови високого рівня.

Мови низького (машинного) *рівня* – асемблери, близькі за структурою до інструкцій процесора. Програмування на таких мовах передбачає знання архітектури та особливостей функціонування комп’ютера. Дозволяє отримати

високоефективні за швидкістю виконання та використання ресурсів комп’ютера програмні продукти.

Мови високого рівня не орієнтовані на апаратні засоби, зручні для програміста, але програмний продукт як правило використовує надлишкові ресурси комп’ютера.

Програми, створені мовою високого рівня, переводяться на машинну мову інтерпретаторами та компіляторами. Інтерпретатор автоматично у міру введення програми перетворює її команди на команди машинної мови. А компілятор транслює (переводить) усю введену програму за командою програміста. Результатом компілювання є згенерований об’єктний код, який і виконується безпосередньо комп’ютером. Скомпільований варіант програми можна зберігати на диску. Для повторного виконання програми компілятор уже не потрібен, оскільки достатньо завантажити з диска в пам'ять комп'ютера скомпільований перед цим варіант і виконати його.

Системи програмування надають зручний інтерфейс для створювання та налагоджування програм тією чи іншою мовою і включають наступний мінімум засобів розробки програм: редактори, транслятори, компонувальники, бібліотеки стандартних програм, системи відлагодження, системи візуалізації та різні сервісні можливості.

Словник. Обчислювальна система, апаратне забезпечення, програмне забезпечення, принципи фон Неймана, архитектура, процесор, пам'ять, принцип адресності, система команд процесора, машинна команда, код операції, операнд, змінна, операційна система, файл, файлова система.

Завдання на СРС: Схема роботи процесора для триадресної моделі ЕОМ [1,с.13]

Контрольні запитання

1. Навести складові обчислювальної системи та дати їм характеристику
2. У чому полягають принципи фон Неймана і яке відображення вони мають у сучасних комп'ютерах?
3. Навести основні відміни різних типів пам'яті, представлених у комп'ютерах.

4. Що таке система команд процесора?
5. Що відбувається за один такт роботи процесора?
6. Навести складові програмного забезпечення та дати їм характеристику.
7. Які етапи містить процес утворення програми?
8. Які різновиди програмного забезпечення можна віднести до прикладного?

Лекція №2

Тема лекції: Форми збереження інформації у комп’ютері. Представлення цілих чисел у пам'яті комп’ютера

План лекції

1. Системи числення. Основні визначення
2. Розгорнута і згорнута форми запису числа
3. Правила переведення чисел у десяткову та із десяткової системи числення
4. Переведення чисел із системи числення з основою p у систему числення з основою q
5. Зв'язок між системами числення з основою 2^k

Зміст лекції

2.1 Системи числення. Основні визначення

Системою числення називають сукупність правил та прийомів запису чисел. Для запису чисел використовують знаки, які називають *цифрами*, а запис власне числа, що складається із цифр, називають *кодом числа*.

Розрізнюють позиційні і непозиційні системи числення (СЧ).

Непозиційними називають системи числення, у яких кількісний еквівалент цифри не залежить від її положення в коді числа.

Непозиційні системи числення історично існували раніше за позиційні, прикладом може слугувати римська система числення, яку використовують і в наші дні.

Наприклад, число 80 у римській СЧ має вигляд LXXX, а цифра X означає 10 незалежно від її положення в коді числа.

Позиційними називають системи числення, у яких кількісний еквівалент цифри залежить від її положення в коді числа.

Наприклад, у звичній для нас десятковій СЧ цифра 3 у числі 333 означає і 3 сотні, і 3 десятка і 3 одиниці.

Для кожної позиційної СЧ основною характеристикою є основа або базис.

Основа СЧ – це кількість різних цифр, за допомогою яких можна представити будь-яке число даної СЧ.

Сам набір цифр називають *алфавітом* даної СЧ, а кількість цифр у коді числа – його *розрядністю*.

2.2 Розгорнута і згорнута форми запису числа

Основа десяткової СЧ – 10, для зображення десяткового числа використовують десять арабських цифр – від 0 до 9, а, наприклад, десяткове число 256 має розрядність три. Це число може бути представлене так:

$256=200 + 50 + 6 = 2*100 + 5*10 + 6 = 2*10^2 + 5*10^1 + 6*10^0$.

Тут *код числа* або запис числа у *згорнутій формі* – 256, а у *розгорнутій формі* запис числа надається по степенях числа 10 – основи нашої СЧ.

У комп'ютерах використовують систему з основою два – двійкову СЧ, а для спрощення представлення двійкових чисел для програмістів – ще й системи з основою 8 та 16.

У загальному випадку число у будь-якій СЧ у розгорнутій формі можна записати так:

$$A_q = a_n * q^n + a_{n-1}q^{n-1} + \ldots + a_0 q^0 + a_{-1} q^{-1} + a_{-2}q^{-2} + \ldots + a_{-m}q^{-m}$$

Тут Aq – код числа, q – основа СЧ, aj – j-та цифра в коді числа, n і m – кількість розрядів цілої і дробової частини відповідно.

Наприклад:

$$1010{,}1101_2 = 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 + 1\cdot 2^{-1} + 1\cdot 2^{-2} + 0\cdot 2^{-3} + 1\cdot 2^{-4}$$

Цифра, що позначає основу, відсутня як окремий символ. Тобто для десяткової СЧ «десять» – це число, а не цифра і відповідно принципу позиційної системи має бути позначено як одиниця з наступним нулем.

У загальному випадку, можна сказати, що для системи з основою q число q потрібно позначати як 10.

Для того, щоб підкреслити, що число задане саме у десятковій системі пишуть 256_{10} .

У загальному випадку будь-яке число можна представити у будь-якій системі числення і алфавіт для неї можна задати також за любим принципом.

Але як видно з наведених прикладів, загальновживаним є такий принцип, що для СЧ з основою $q<10$ алфавіт складають десяткові цифри від 0 до $q-1$, а

з основою *q>10* – десяткові цифри від *0* до *9* і латинські букви у порядку абетки.

Тобто для двійкової системи це два числа – 0 і 1, для системи з основою 5 – 0, 1, 2, 3, 4, а для шіснадцяткової системи – це символи від *0* до *9* і від *A* до *F*.

Наприклад, число $10_{10}=1010_2=20_5=A_{16}$

2.3 Правила переведення чисел у десяткову та із десяткової системи числення

Переведення у десяткову СЧ. Якщо число, представлене у будь-якій СЧ, записати у розгорнутій формі, а потім виконати дії у десятковій СЧ, це приведе до переведення цього числа у десяткову СЧ.

Для прикладу можна розглянути представлення одного і того ж числа у різних СЧ, записане у згорнутій і розгорнутій формах, виконати дії у десятковій СЧ для переведення даного числа у десяткову СЧ.

$1111000_2=1*2^6+1*2^5+1*2^4+1*2^3+0*2^2+0*2^1+0*2^0= 64+32+16+8+0+0+0=120_{10}$

$170_8=1*8^2+7*8^1+0*8^0=64+56=120_{10}$

Приклад переведення числа, яке має дробову частину.

$100000,11_2=1*2^5+0*2^4+0*2^3+0*2^2+0*2^1+0*2^0+1*2^{-1}+1*2^{-2}=32+1/2+1/4=32,75_{10}$

$40,3_8=4*8^1+0*8^0+3*8^{-1}=32+3/8=32,75_{10}$

$1012,202_3=1*3^3+0*3^2+1*3^1+2*3^0+2*3^{-1}+0*3^{-2}+2*3^{-3} = 27+3+2+ 2/3+ 2/27+\ldots\approx32,737_{10}$

Переведення із десяткової СЧ. Для зворотного перетворення необхідно виконати зворотні дії – ділити десяткове число на основу потрібної СЧ.

Для цілої частини числа алгоритм наступний. Після першого ділення числа A_{10} на нову основу q, отримаємо остачу, яка відповідатиме молодшому розряду числа в системі з основою q. Якщо частку знову поділити на q, остача відповідатиме другому з кінця розряду у новій системі. Такі ділення потрібно продовжити доти, доки не буде отримана частка, менша за основу q. Зрозуміло, що запис числа в A_q потрібно формувати із остач кожного ділення, а у першому розряді стоятиме остання частка.

Приклад переведення цілого чисел діленням.

124_{10} ->A_2 = 1111100_2

```
 124 |  2
 124 | 62 |  2
   0   62| 31|  2
       0  30| 15|  2
          1  14|  7 |  2
             1   6 |  3| 2
                 1   2| 1
                      1
   ←--------------------┘
        1111100₂
```

Алгоритм переведення дробових частин також інтуїтивно зрозумілий: кожний черговий розряд числа отримуємо множенням на основу отриманої в результаті попереднього множення дробової частини і врахуванням отриманої цілої частини добутку.

Приклад переведення дробового числа множенням.

0.125_{10} -> A_2 = 0.001_2

```
  0 | 125
    |  2
  0 | 250
    |  2
  0 | 500
    |  2
  1 | 000
```

2.4 Переведення чисел із СЧ з основою *p* у СЧ з основою *q*

Розглянуті алгоритми працюватимуть при переведенні із будь-якої системи і у будь-яку систему. Потрібно просто дії виконувати у тій системі, до якої заплановано переводити.

Якщо переведення потрібно робити вручну і із не десяткової СЧ у не десяткову, зручно користуватися так званою схемою Горнера.

Ідея. Щоб перевести число із СЧ з основою q до СЧ з основою p, потрібно записати число A_q у розгорнутій формі в позначеннях СЧ p і виконувати дії у цій системи. Для зручності усних обчислень роблять нескладні перетворення:

$$A_q = a_n*q^n + a_{n-1}q^{n-1} + \dots + a_0q^0 = ((\dots(a_n*q + a_{n-1})q + a_{n-2})q + \dots + a_0q^0$$

Наприклад A_2 -> A_3

$10100_2 = 1*2^4+0*2^3+1*2^2+0*2^1+0*2^0 = (((1*2+0)*2+1)*2+0)*2+0 = 202_3$

Для перевірки можна обидва числа перевести в A_{10}, вийде 20 в обох випадках, отже переведення виконано правильно.

2.1 Зв'язок між системами числення з основою 2^k

Для переводу цілого числа з двійкової системи до вісімкової його попередньо треба розбити на групи по три (тріади), а потім кожну тріаду замінити відповідним цифрою вісімкової системи:

11001101111011_2=11 001 101 111 011_2=31567_8
3 1 5 7 3

Щоб перейти до шістнадцяткової системи треба зробити так само, тільки поділяти потрібно не на тріади, а на на групи по чотири – тетради, і замінювати їх шістнадцятковою цифрою:

11001101111011_2= 11 0011 0111 1011_2 =$337B_{16}$
3 3 7 B

Зрозуміло, що для зворотної дії потрібно діяти навпаки. Кожну цифру вісімкового числа замінити тріадою двійкових цифр, а шіснадцяткового – тетрадою. Наприклад:

1204_8=001 010 000 100_2 = 1010000100_2
$70AE_{16}$= 0111 0000 1010 1110_{16} = 111000010101110_{16}

Потрібно звернути увагу на те, що нулі на початку числа можна відкинути, число від цього не зміниться, а от в середині і в кінці числа цього робити, звичайно не можна.

Так само діємо і з дробовими числами.

Наприклад:

11001101000.01_2=11 001 101 000. 010_2=3150.2_8
11001101000.01_2=110 0110 1000. 0100_2=668.4_{16}

Такий підхід до переведень можливий тому, що 8 і 16 є степенями 2 і для представлення будь-якої цифри цих систем необхідно і достатньо відповідно трьох або чотирьох двійкових розрядів.

Словник. Система числення (СЧ), позиційна СЧ, основа СЧ, алгоритми переведення в СЧ: ділення, множення, схема Горнера, тетрада, тріада.

Контрольні запитання

1. Що таке система числення?

2. Чим відрізняється позиційна і непозиційна системи числення?
3. Як перевести число із двійкової до десяткової системи числення?
4. Як перевести ціле число із десяткової до двійкової системи числення?
5. Як перевести дріб із десяткової до двійкової системи числення?
6. Який принцип запису числа з основою більше 10?
7. З використанням яких систем числення інформація представлена в комп'ютері і чому?
8. Які системи числення використовують в програмуванні і чому?

Контрольні завдання

1. Записати число $110011,11_2$ у розгорнутій формі, перевести в СЧ з основою 10.
2. Перевести число 265 у системи числення з основою 2, 8, 16, 3.
3. Перевести число 1100112 у системи числення з основою 3, 8 та 16 напряму, виконати перевірку.

Лекція №3

Тема лекції: Форми збереження інформації у комп'ютері. Внутрішнє представлення різнотипної інформації

План лекції

1. Арифметика у різних системах числення
2. Представлення чисел з фіксованою крапкою у пам'яті комп'ютера
3. Представлення чисел з плаваючою крапкою у пам'яті комп'ютера

Зміст лекції

3.1 Арифметика у різних системах числення

Табличка відповідностей цифр у десятковій, двійковій, вісімковій і шіснадцятковій системах числення наступна:

Основа 10	**2**	**8**	**16**
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Двійкова система числення

Таблиця додавання

+	0	1
0	0	1
1	1	10

Таблиця множення

×	0	1
0	0	0
1	0	1

Обчислення проводять у стовпчик, так само, як і в десятковій СЧ. Якщо число не ділиться націло, домовляються про кількість знаків після коми.

Приклади:

10001 + 10100 100101	11001 + 10111 110000	11111 + 1 100000	1010,011 + 11,010 1101,101
10101 – 00011 10010	100000,01 – 1 11110,01	1101,1 × 11 11011 11011 101000,1	11001|101 101 |101 101 101 0

Вісімкова система числення

Таблиця додавання

+	1	2	3	4	5	6	7
1	2	3	4	5	6	7	10
2	3	4	5	6	7	10	11
3	4	5	6	7	10	11	12
4	5	6	7	10	11	12	13
5	6	7	10	11	12	13	14
6	7	10	11	12	13	14	15
7	10	11	12	13	14	15	16

Таблиця множення

×	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	10	12	14	16
3	3	6	11	14	17	22	25
4	4	10	14	20	24	30	34
5	5	12	17	24	31	36	43
6	6	14	22	30	36	44	52
7	7	16	25	34	43	52	61

Приклади обчислень у вісімковій системі:

34,24 + 45,36 101,62	36,34 – 17,43 16,71	15,21 × 4,1 1521 6304 64,561	2654|13 26 |204 54 54 0

3.2 Представлення чисел з фіксованою крапкою у пам’яті комп’ютера

У комп’ютері використовують дві форми запису чисел – з фіксованою і з плаваючою крапкою.

У записі числа з фіксованою крапкою положення крапки заздалегідь визначене і однакове для будь-якого числа, наприклад, після молодшої цифри його двійкового запису. Тоді дробова частина відсутня і крапка у зображенні числа не потрібна. Так можуть зберігатися цілі числа.

Для визначення знаку числа використовують перший розряд: 0 – число додатне, 1 – від’ємне.

Для кодування чисел використовують три різні системи кодування. Кодування додатних чисел – однакове у будь-якій системі кодування. Для

кодування від'ємних чисел використовують прямий, обернений і додатковий коди. В мові С для запису цілого числа використовують типи *int, short int, long int,* яким відповідають певні формати представлення цілих чисел – один, два або чотири байти (залежно від компілятора).

Без втрати загального підходу, розглядатимемо представлення цілих чисел як однобайтних (тобто в межах від -127 до $+127_{10}$).

Представлення чисел у *прямому* коді:

+12 – 0 0001100 (або 0 0000000 00001100 для слова)
- 12 – 1 0001100 (1 0000000 00001100 для слова)

Представлення додатних чисел в *оберненому* коді таке саме, як і в прямому, а для від'ємних чисел кожний розряд , крім знакового, інвертують, наприклад:

+12 – 0 0001100 (або 0 0000000 00001100 для слова)
12 – 1 1110011 (або 1 1111111 1111 0011 для слова)

Представлення числа нуль в оберненому коді може бути як +0 і тоді це 00000000, а може бути як -0 і тоді це 11111111.

Представлення чисел у *додатковому* коді незмінне для додатного числа, а представлення від'ємного формують додаванням одиниці до молодшого розряду представлення цього числа у оберненому коді, наприклад:

+12 – 0 0001100 (або 0 0000000 00001100 для слова)
12 – 1 1110100 (або 1 1111111 1111 0100 для слова)

Представлення числа нуль у додатковому коді може бути тільки одне – це 00000000.

(+12)+(-12)=000011002+111101002=000000002=010

Використання оберненого і додаткового кодів спрощує можливість автоматизації процесу віднімання двох чисел. Оскільки віднімання замінено за таких умов додаванням від'ємного числа.

Віднімання в оберненому коді – це додавання додатного і від'ємного числа. Наприклад:

4210-3610 : 42 = 0 0101010пр, -36 = 1 0100100пр, 1 1011011зв
0 0101010 + 1 1011011= 0 0000110
1 1011011 Якщо відбувається переповнення розрядної сітки
---------- як у наведеному прикладі, тоді одиничку переповнення
1 0 0000101 додають до молодшого розряду результату.
↓ 1
↓_ _____ _↑
0 0000110

Віднімання у додатковому коді – це також додавання додатного і від'ємного числа, але необхідність у додаванні одиниці у випадку переповнення відпадає. Наприклад:

42_{10}-36_{10} : 42 = 0 $0101010_{пр}$, -36 = 1 $0100100_{пр}$ = 1 $1011011_{зв}$ = 1 $1011100_{дод}$

```
  0 0101010 + 1 1011100= 0 0000110
  1 1011100
 ----------
 10 0000110
←┘
```

У сучасних комп'ютерах цілі числа представлені у додатковому коді.

3.3 Представлення чисел з плаваючою крапкою у пам'яті комп'ютера

У записі числа з плаваючою крапкою – положення крапки не зафіксоване і кожного разу повинно задаватися окремо, тоді запис одного числа може приймати різний вигляд в залежності від обмежень, що накладаються на форму.

У загальному випадку число *A* у системі числення з основою *q* можна представити у експоненційній формі. Тобто у такому вигляді:

$$A_q = m \cdot q^n,$$

де *m* – мантиса, *n* – порядок числа. Якщо мантиса задовольняє співвідношенню $1 \le |m| \le q$, то число нормалізоване.

Залежно від апаратної реалізації дійсні числа, оголошені в С як ***float*, *double*, *long double***, можуть займати 32, 64, 80 бітів пам'яті. Тоді вони матимуть відповідну структуру, наприклад для *float:*

Знак	Характеристика	Нормалізована мантиса
31	30 23	22 0

Нормалізована мантиса – це мантиса без першого значущого числа (тобто без першої одиниці) у двійковому записі числа. Тому запис мантиси у пам'ять починають з другої цифри. Будь-яке перетворення числа під час обчислень приведе до автоматичного відновлення першої одиниці.

Порядок може приймати значення від -127 до +127. Щоб знак порядку явно не зберігати, його записують як завжди додатний, збільшеним на 127. Це називається зміщеним порядком або *характеристикою.*

Наприклад: Визначити машинне представлення числа 42.25 як 32-бітного числа з плаваючою крапкою.

Двійкове представлення числа: 101010. 01 -> 1.01010 01*2^5

0101001 – нормалізована мантиса

$5_{10} = 101_2$ -> 101+1111111=10000100

Знак + -> перший розряд 0

Тоді: 0 10000100 01010010000000000000000

Словник. Біт, байт, слово, подвійне слово, розрядність процесора, прямий, обернений, додатковий код, представлення з фіксованою крапкою, представлення з плаваючою крапкою, мантиса, порядок, нормалізація числа.

Завдання на СРС: Розібрати представлення числа з плаваючою крапкою для 16- та 64-бітного слова [1,с.31, 2,с.15]

Контрольні запитання та контрольні завдання

1. Які системи кодування використовують у сучасних комп'ютерах?
2. Чим відрізняються алгоритми віднімання для прямого, оберненого і додаткового кодів?
3. Що таке мантиса, порядок?
4. Як обчислити мантису і порядок нормалізованого числа?
5. Виконати дії у відповідній системі числення, перевірити в десятковій
   а)110111_2*1001_2 б)1011001_2:1100_2 – до двох знаків після коми
6. Виконати дії у відповідній системі числення
   а)731_8-256_8 731_8+256_8
   б) $3FC_{16}$+$82A_{16}$ $E1C_{16}$-$A32_{16}$
7. Виконати дії у прямому, оберненому та додатковому кодах
   а)101011 – 111 б)100011 – 110000
8. Яке машинне представлення матиме число -31,625 як 32-бітне дійсне?
9. Якому десятковому числу відповідає двійковий код 01000001 10100000 00000000 00000000?

Розділ 2. Базові елементи та конструкції мови

Лекція №4

Тема лекції: Складові алгоритмічної мови. Елементи мови С

План лекції

1. Склад алгоритмічної мови
2. Алфавіт мови
3. Лексеми
4. Вирази, операнди, змінні і константи

Зміст лекції

4.1 Склад алгоритмічної мови.

Будь-яка мова може бути визначена такими основними компонентами: символами, словами, словосполученнями та реченнями. В алгоритмічній мові їм відповідають: *символи, лексеми, вирази та оператори.* Оператори мови складають *програму*.

Символи – це основні неподільні елементи мови. Представляють собою знаки, які складають усі конструкції мови.

4.2 Алфавіт мови.

Символи утворюють *алфавіт* мови. До алфавіту мови С входять символи, визначені в таблиці кодування ASCII, а саме:

1. Великі та малі латинські літери (розрізняються компілятором).
2. Цифри 0-9.
3. Знаки пунктуації
4. Деякі інші символи

Таблиця 1. Знаки пунктуації

Символ	Назва	Символ	Назва
,	кома	)	права кругла дужка
.	крапка	(	ліва кругла дужка
;	крапка з комою	{	права фігурна дужка
:	двокрапка	}	ліва фігурна дужка
?	знак питання	<	менше
'	апостроф	>	більше
!	знак оклику	]	права квадратна дужка
|	вертикальна риса	[	ліва квадратна дужка

/	дробова риса (слеш)	**#**	решітка
****	зворотна риса (зворотний слеш)	**%**	процент
~	тільда	**&**	амперсанд
*****	зірочка	**^**	виключне «або»
+	плюс	**=**	дорівнює
-	мінус	**"**	лапки
_	знак підкреслення		

4.3 Лексеми.

Лексема – мінімальна одиниця мови, яка має власний зміст та обробляється транслятором. Лексеми формують *вирази*. *Програма* розглядається як послідовність виразів і лексем.

У мові С кожна лексема належить одній з наступних категорій:

- зарезервовані (ключові) слова;
- ідентифікатори;
- літерали;
- знаки операцій;
- розділювачі.

Зарезервовані слова стандарту С89:

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

Зарезервовані слова стандарту С99 (додатково):

```
_Bool               _Imaginery          restrict
_Complex            inline
```

Ідентифікатор – це лексема, яка складена із літер латинського алфавіту, цифр, символів «_»(підкреслення) та «$» (знак долару), які даються програмним об’єктам, тобто змінним, іменованим константам, типам та функціям. Інші символи, крім перелічених, в ідентифікаторах недопустимі. Великі і малі букви в ідентифікаторах відрізняються, ідентифікатор не може співпадати з зарезервованим словом і не може містити пропуск.

Літерали – це значення констант різних типів, які зустрічаються у тексті програми. Зовнішній вигляд літералів залежить від типу, до якого вони

належать. Наприклад, 23 – ціле *int*, 45.6 або 0.5e-3 – дійсні *double, 'g'* – символ *char, "some information"* – рядковий літерал.

Знаки операцій побудовані на основі набору спеціальних символів і букв алфавіту.

Таблиця 2. Знаки операцій

,	!	!=	|	|=	%	%=	&	&&
&=	()	*	*=	+	++	+=	-	--
-=	->	->*	.	.*	/	/=	<	<<
<=	<<=	>	>>	>=	>>=	==	?:	[]
^	^=	~	||	#	##	sizeof		

Розділювачі – це ';' крапка з комою, *'{}'* фігурні дужки , *'...'* трикрапка.

Множина операцій та розділювачів фіксована. Лексеми відокремлюються так званими пробільними символами, кількість яких необмежена. Це власне пропуск, *ENTER*, знаки табуляції.

Інші символи можуть зустрічатись лише у коментарях чи текстових виразах. Багаторядковий коментар починається з символів /* і закінчується символами */, однорядковий коментар починається з символів // і закінчується в кінці рядка. Наприклад:

```
/* це багаторядковий
коментар*/
// це однорядковий коментар
```

4.4 Вирази, операнди, змінні і константи

Вираз задає правило обчислення деякого значення і побудований на основі множини символів операцій, ключових слів та *операндів*. Порядок обчислення виразів визначається пріоритетами операцій, дужками і правилами асоціативності.

Операндами у виразі можуть бути як літерали (або літеральні константи), так і змінні.

Змінна – це іменована область пам'яті, до якої є доступ із програми.

Кожна змінна С має певний *тип*, який характеризує розмір і розташування цієї області пам'яті, діапазон значень, які вона може зберігати, і набір операцій,

застосовних до цієї змінної. Змінна, як і літерал, зберігає своє значення в деякій області пам'яті. Але, на відміну від літерала, до неї можна звернутися за адресою в пам'яті. Зі змінною асоційовані дві величини:

– власне значення, або *r-значення* (*read value*), яке зберігається в цій області пам'яті і властиве як змінній, так і літералу;

– значення адреси області пам'яті, асоційованої зі змінною, або *l-значення* (*location value*) – місце, де зберігається r-значення, притаманне тільки змінній і не визначене для літералу.

Оголошення змінних в програмі може знаходитись

– на зовнішньому рівні, поза усіх функцій, це глобальні змінні;
– в середині функції – локальні змінні;
– у визначенні параметрів функції – формальні параметри.

Загальний вигляд оголошення змінних:

```
[клас_пам'яті][ розширення_типу] тип список_змінних;
```

де **список_змінних** можна представити так:

```
змінна[ініціалізатор][,список змінних];
```

Наприклад:

```
int student_count, i=0,j=1;
double slr = 9999.99, wage = slr + 0.01;
```

Змінна, оголошена на глобальному рівні, одразу буде ініціалізована нульовим значенням відповідно типу. Змінна, оголошена на локальному рівні або у динамічній пам'яті, може містити деяке випадкове значення.

Словник. Алфавіт, символ, слово, лексема, вираз, оператор, ідентифікатор, спеціальний символ, знак операції, пробільні символи, розділовий знак, ключове слово, зарезервоване слово, змінна, константа, тип, літерал.

Завдання на СРС: Історія створення, початкове призначення та причини поширення мови С [4, с.10]

Контрольні запитання

1. З яких частин складається алфавіт мови С?
2. Що таке лексема? Які існують різновиди лексем в С?
3. Що таке ідентифікатор? За якими правилами утворюються ідентифікатори?

4. Що таке знаки операцій і як вони утворюються?
5. Що таке ключове слово?
6. Що таке літерал? Які різновиди літералів і правила їх утворення?
7. Що таке вираз? Операнд? Змінна?
8. Що таке r-value і l-value?

Лекція №5

Тема лекції: Структура, оформлення, етапи виконання С- програми

План лекції

1. Загальна структура програми
2. Директиви препроцесора
3. Оформлення функцій
4. Етапи виконання програми

Зміст лекції

5.1 Загальна структура програми

Програма – сукупність операторів однієї алгоритмічної мови, об'єднаних деяким алгоритмом.

Програма мовою С складається з директив препроцесора, зовнішніх описів і функцій і має таку структуру:

- директиви препроцесора
- описи
- функції

У будь-якому місці програма може мати коментарі.

Програма мовою С записується у текстовий файл з розширенням `.c`, мовою С++– з розширенням `.cpp` з використанням будь-якого текстового редактора, наприклад, **`prog1.cpp`.**

Директиви препроцесора вказують на перетворення, які потрібно зробити з текстом програми. За їх допомогою також здійснюється оголошення іменованих констант, що використовуються у всій програмі, підключення інформації про стандартну бібліотеку та ін.

В розділі *описів* роблять глобальні оголошення. Можуть містити оголошення будь-яких даних – змінних, іменованих констант, користувацьких типів, функцій. Оскільки вони розташовані на зовнішньому рівні – вони глобальні.

Функції – це основні складові програми. Представляють собою синтаксично та семантично завершений самостійний фрагмент програми, призначений для розв'язання певної задачі. Складаються із послідовності операторів у фігурних дужках, кожний з яких відокремлений символом **';'**.

Програма може містити декілька функцій, а може – лише одну, але обов'язково функцію з назвою **main().**

Приклад 1. Проста програма мовою С:

```
#include <stdio.h>              //директива препроцесора
int main(){                     //оголошення функції
     printf("hello, world\n");//виклик функції виведення
 return 0;
}
```

Виконання програми завжди починається з функції **main().** Із функції **main()** для виконання різних операцій можуть бути викликані інші функції, які або знаходяться у стандартних бібліотеках, або мають бути написані самим програмістом.

5.2 Директиви препроцесора

Препроцесор — це складова частина компілятора, яка проводить попередню обробку програми. Директиви записуються в окремих рядках і починаються символом **#.** Після директиви розділові знаки не ставлять.

Перший рядок розглянутої програми містить директиву **#include**:

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

– це директива препроцесора, яка вказує на необхідність додати до програми інформацію про стандартну бібліотеку. У даному випадку, `stdio.h` – заголовковий файл, що містить оголошення функцій уведення-виведення, у тому числі і функцію **printf** із наведеного прикладу.

Директива **#define,** наприклад, виконує *макропідстановку* або *макрозаміну* у всьому тексті програми. Нею користуються, наприклад, для заміни *макроконстанти* на значення деякої константи по всьому тексту програми, як показано у наступному прикладі.

Приклад 2. Програма з використанням макропідстановок.

```
#include <stdio.h>
#define RAD_PI 3.1419265
#define DIG_PI 180
int main()
{
    double angle_r, angle_d;
    printf("enter angle in radians\n");
    scanf("%lf", &angle_r);
    angle_d=angle_r*DIG_PI/RAD_PI;
    printf("angle in degrees=%lf\n",angle_d);
```

```
    printf("enter angle in degrees\n");
    scanf("%lf", &angle_d);
    angle_r=angle_d*RAD_PI/DIG_PI;
    printf("angle in radians=%lf\n",angle_r);
return 0;
}
```

У даному тексті використано дві макроконстанти – **RAD_PI** та **DIG_PI**.

5.3 Оформлення функції

Опис функції містить заголовок і розташоване у фігурних дужках тіло функції. Синтаксис опису функції:

```
тип_що_повертає ідентифікатор ( [список_параметрів])
{
оператори_функції;
};
```

Заголовок функції містить інформацію, яка однозначно її оголошує для того, щоб вона могла бути викликана із іншої функції.

Один з методів передачі даних між функціями полягає в тому, що одна функція викликає іншу і надає їй список параметрів – змінних, які називають *аргументами*. Цей список розташований у круглих дужках одразу після імені функції. У прикладі функція `main` викликає `printf` з аргументом `"hello, world\n"`. Функція `printf` виводить свої аргументи на екран або на друк. Функція `main` не має аргументів, на що вказує порожній список в дужках : `int main()`.

Тіло функції складається з послідовності операторів, розділених знаком ‘;’.

Оператори бувають виконуваними – такими, що задають дії над даними – і невиконуваними – такими, що задають описи даних.

Оператори можуть бути *прості*, навіть порожні – із одного розділового знаку ‘;’, а можуть бути *складені*. Послідовності операторів можуть утворювати *блоки*, розташовані у окремих фігурних дужках.

Функція `main()` може нічого не повертати в ОС, яка її запустила, а може повертати ціле `int`, яке містить код завершення – нормально (0) чи аварійно. В обох прикладах функції повертали 0 в останньому виконуваному операторі `return 0;`

5.4 Етапи виконання програми

Усі сучасні системи програмування мають свої текстові редактори, які дозволять легко і зручно уводити текст програми, часто одразу включені засоби оперативного редагування. Ключові слова, лексеми, ідентифікатори, коментарі одразу будуть розпізнані і виділені кожний своїм кольором, використання ідентифікаторів функцій приводить до активізації підказок про кількість та типи параметрів, тощо.

До початку компіляції програму обробляє препроцесор – послідовно виконує усі директиви. Доповнює текст програми заголовками функцій із потрібних файлів за допомогою `include`, робить макрозаміни та макропідстановки по всьому тілу програми, визначені директивами `define`, вилучає коментарі, тощо.

Підготовлений препроцесором повний текст програми обробляє компілятор. Компілятор перевіряє програму на наявність синтаксичних помилок. Якщо вони є, видає відповідне повідомлення і припиняє роботу, якщо немає, запускає процес компіляції і формує файл з так званим об'єктним кодом – набором двійкових інструкцій – машинних команд, які може виконати комп'ютер. Такий файл має те саме ім'я і розширення **`.obj`**.

На етапі компоновки до об'єктного коду програми редактор зв'язків прив'язує об'єктні коди потрібних бібліотечних функцій і створює так званий виконуваний код програми у вигляді файлу з тією самою назвою і розширенням `.exe`.

Цей файл уже не потребує середовища програмування і його можна запускати просто із операційного середовища.

Словник. Препроцесор, директива препроцесора, системна бібліотека, стандарт мови, заголовковий файл, глобальні оголошення, функція, прототип функції, головна функція, компіляція, транслятор, компілятор, компоновка, виконання, текстовий файл, об'єктний файл, виконуваний файл, середовище програмування.

Завдання на СРС*:* Директиви препроцесора. Директива `#include` [4,с.229, 3,с.93]

Контрольні запитання

1. Назвіть основні складові коду програми.
2. Що таке функція?
3. Що називається заголовком і тілом функції.
4. Для чого потрібен і як працює препроцесор?
5. Що таке коментар? Які види коментарів існують в мові С?
6. Назвіть етапи утворення виконуваного коду програми.

Лекція №6

Тема лекції: Типи даних. Організація уведення-виведення

План лекції

1. Типи даних
2. Специфікатори рядка формату
3. Форматне виведення даних
4. Форматне введення даних

Зміст лекції

6.1 Типи даних

Тип даних - це множина допустимих значень, які може приймати той чи інший об'єкт. Тип залежить від внутрішнього представлення інформації, що визначає як саме зберігаються й обробляються дані даного типу.

Тип даних визначає:

• внутрішнє представлення даних в пам'яті комп'ютера;

• обсяг пам'яті, що виділяється під дані;

• діапазон значень, які можуть приймати величини цього типу;

• операції та функції, які можна застосовувати до даних цього типу.

Стандарт С89 визначає п’ять фундаментальних типів, на основі яких формують інші типи. Це:

`char, int, float, double, void.`

Залежно від компіляторів і процесорів діапазон значень цих типів може бути різним, але: **`char`** – завжди байт, **`int`** – завжди слово (16 бітів на 16-розрядному і 32 – на 32-розрядному процесорах, але також не завжди). В С++ до цих типів додаються **`wchar_t, bool`**, у С99 - **`_Bool, _Complex.`**

Стандарт визначає тільки мінімальний діапазон значень кожного типу, але не розмір у байтах.

Тип **`void`** має порожню множину значень, його використовують для оголошення функції, яка не повертає значення, для створення порожнього списку аргументів, для створення універсального вказівника, в операціях приведення типу.

Також для створення типу можуть бути використані модифікатори (специфікатори, описувачі) типу, які передують опису базового типу в оголошеннях. Це:

signed, unsigned – визначають наявність знаку;

long, short – визначають розмір.

Для **char** використовують тільки два перші, для решти типів – усі, при цьому можуть бути використані сполучення одного з перших з одним із останніх.

Повна інформація про типи C89(C99) представлена у додатку А.

Цілі типи. Базовими цілими типами в С є типи **int** та **char.** До цих типів можуть бути застосовані усі перелічені модифікатори.

Приклад оголошення, де оголошення подані як переліки і деякі зі змінних одразу ініціалізовані.

```
int a, b=23, c;
long int len=-40000, big;
unsigned u=45;
```

Якщо в оголошенні заданий модифікатор без типу, то за умовчанням це **int**. До даних цілого типу можуть бути застосовані наступні *арифметичні операції :*

+	додавання
-	віднімання, також унарний мінус
*****	множення
/	цілочисельне ділення: 13/3 = 4
%	остача від цілочисельного ділення: 13%3 = 1
=	оператор присвоювання
++x;	префіксний інкремент x=x+1
x++;	постфіксний інкремент x=x+1
--x;	префіксний декремент x=x-1
x--;	постфіксний декремент x=x-1

Порозрядні операції застосовують тільки до даних цілого типу і вони визначають дію безпосередньо над бітами чисел:

&	бінарна операція ,побітове *І*
|	бінарна операція , побітове *АБО*
^	бінарна операція , виключне *АБО*
~	унарна операція *НЕ* – заперечення, доповнення до 1
>>	бінарна операція ,зсув вправо, в напрямку від старшого біта

<< бінарна операція ,зсув вліво, в напрямку від молодшого біта

Дійсні типи. Для зберігання дійсних чисел застосовуються типи даних **float** (з одинарною точністю) і **double** (з подвійною точністю). Формат зберігання – з плаваючою крапкою.

Приклади оголошення

```
double result, some=4.5, eps=.1e-8;
long double p, p1=-35.67891234005;
```

До даних дійсного типу можуть бути застосовані наступні арифметичні операції :

+	додавання
-	віднімання, також унарний мінус
*****	множення
/	ділення: 15/5 = 7.5

Для застосування математичних функцій до даних числових типів, потрібно підключити заголовковий файл з математичними функціями **#include <math.h>.** Деякі функції, оголошені у цьому файлі, потрібні для виконання першої роботи комп'ютерного практикуму, представлені у додатку Б.

Дані з плаваючою крапкою можна привести до цілого типу округленням дійсного аргументу x до найближчого цілого:

```
long int lround(double x);
```

Символьні типи. У стандарті С немає типу даних, який можна було б вважати дійсно символьним. Для представлення символьної інформації використовують тип **char.** Змінна такого типу призначена для зберігання одного символу. Приклад оголошення з ініціалізацією.

```
char c='A', a,b='5';
```

Оскільки в пам'яті комп'ютера символи зберігаються у вигляді їх ASCII-кодів – цілих чисел, то тип **char** насправді є підмножиною типу **int.** Під величину символьного типу відводиться 1 байт.

Для зберігання інформації у вигляді рядка, використовують масиви із елементів типу **char.**

Логічний тип. У стандарті С89 немає окремо визначеного логічного типу. Тому у разі потреби його заміняє тип **`int`** зі значеннями **`0`** і **`1.`** У пізніших стандартах С і С++ з'явився спеціальний булевий тип **`bool`** зі значеннями **`true(1)`** та **`false(0).`**

6.2 Форматне виведення даних

Мова С не містить вбудованих засобів для забезпечення уведення-виведення даних. Для обміну даними з будь-яким зовнішнім пристроєм потрібно підключити відповідний заголовковий файл `#include <stdio.h>`. Для виведення інформації використовують функцію `printf`, яка має наступне оголошення:

```
int printf(рядок_формату, список_виразів_виведення);
```

Повертає кількість виведених символів (з \0 включно) або від'ємне число з кодом помилки.

Ланцюжок формату містить послідовність символів, яку потрібно вивести на екран разом із специфікаторами формату. Специфікатори формату знаходяться серед символів виводу у тих положеннях, де повинні стояти значення відповідних аргументів із списку виведення.

Кількість, тип та порядок специфікаторів і аргументів повинні співпадати, за це відповідає програміст. Якщо більше специфікаторів – результат невизначений, найчастіше – виведуться випадкові значення. Якщо більше аргументів – «зайві» будуть втрачені – не будуть виведені.

6.3 Специфікатори рядка формату

Специфікації формату даних призначені для управління формою зовнішнього представлення значень аргументів функції **`printf().`**
Узагальнений формат специфікації перетворення має вигляд:

`%[прапорці][ шир_поля.][точність][модифікатор]специфікатор`

`%` – ознака початку специфікації

`Прапорці` керують формою зображення і розташування даного, залежать від специфікатора

`Ширина` – ціле, визначає ширину поля виведення, буде врахована тільки якщо є достатньою для даного параметра, інакше – відповідно умовчання для

даного специфікатора. Вирівнювання – по правому краю, якщо перед значенням ширини – мінус, то по лівому.

Точність визначає кількість символів у рядку виведення для даної змінної, залежить від специфікатора.

Модифікатор – уточнює тип.

Специфікатор – визначає тип даного параметра. Це кінець специфікації для даного параметра.

Серед елементів специфікації перетворення обов'язкові є тільки два - символ '**%**' і **специфікатор.**

Специфікатори виведення:

d та **i**- для цілих десяткових чисел **int**;

f - для дійсних чисел у формі з фіксованою крапкою, **e(E)** - для дійсних чисел у формі з плаваючою крапкою (з мантисою і порядком), **g(G)** – як **f** або **e(E)**- залежно від того, який варіант можна записати коротше – для **float** і **double**;

c – символ; **s** – рядок; **p** – вказівник;

a(A), o, u, x(X) – відповідно у 10-ковому цілі (тільки в С99), цілі у 8-ковому, 10-ковому без знаку, 16-ковому без знаку.

Наприклад після виконання операторів

```
float c, e;
int k;
c=48.3; к=-83; e=16.33;
printf ("c=%f, k=%d, e=%e", c, k, e);
```

на екрані вийде такий рядок:

```
c=48.299999, к=-83, e=1.б3300e+01
```

6.4Форматне введення даних

Функція **scanf()** виконує "читання" кодів, які уводяться з клавіатури.

```
int scanf(рядок_формату,список_адрес_уведення);
```

Повертає кількість успішно зчитаних символів. Це можуть бути коди видимих символів і управляючі коди. Прийняті коди **scanf()** перетворює у внутрішній формат відповідно до специфікацій рядка формату і передає програмі.

Рядок формату і список аргументів для функції **scanf()** обов'язкові. Рядок формату для функції **scanf()** має вигляд:

%[* ширина_поля][модифікатор] специфікатор

Специфікатори ті самі, що і для введення. Відміни такі: для уведення числа типу **double** використовують специфікатор **%lf,** а уведення рядка відбуватиметься тільки до першого пробільного символу.

На відміну від функції **printf()** аргументами для функції **scanf()** можуть бути тільки адреси об'єктів програми. Наприклад:

```
scanf ("%d%lf%f",&n,&z,&x);
```

В даному прикладі специфікації перетворення у форматному рядку не містять відомостей про розміри полів і точність значень, що вводяться.

Словник. Уведення, виведення, консоль, формат, специфікатор формату, форматне введення, форматне виведення, модифікатор, точність, прапорець, ширина поля, вирівнювання.

Контрольні запитання

1. Що таке рядок формату?
2. Що таке специфікації перетворення формату? Перерахувати основні.
3. Яку структуру мають специфікатори у загальному випадку?
4. Чим відрізняються списки введення і виведення даних?

Лекція №7

Тема лекції: Використання керуючих конструкцій мови С

План лекції

1. Класифікація виконуваних операторів
2. Оператори-вирази
3. Проста лінійна програма. Стандартні математичні функції
4. Деякі обчислювальні алгоритми
5. Використання макропідстановок

Зміст лекції

7.1 Класифікація виконуваних операторів

Серед виконуваних операторів виділяють прості і складені (включають до свого складу інші оператори).

До простих операторів відносяться

- оператори-вирази
- умовні оператори
- оператори циклу
- оператори переходу

Окремо можна розглядати порожній оператор і блок.

7.2 Оператори-вирази

До *операторів-виразів* відносять будь-який вираз, який означає дію. Це може бути:

- Виклик функції яка не повертає результату – будь-яка функція, що повертає **`void`**, наприклад **`srand(),`** або виклик без присвоювання (**`scanf(),`** наприклад),
- Вираз типу **`a>MAX ? ++x: ++y;`**
- Оператор присвоювання, якщо справа в операції присвоювання знаходиться деякий вираз, побудований за усіма розглянутими раніше правилами, наприклад:

```
y=sin(2.5*x)/cos(1+x*x);
y+=x;
x++;
a=b=c=5.5;
```

Програму, яка реалізує лінійний алгоритм, складає сукупність послідовно записаних операторів-виразів.

7.3 Проста лінійна програма

Для написання простої лінійної програми, яка має значення як обчислювальна, потрібно розібратися з основними математичними функціями стандартної бібліотеки із заголовкового файлу `<math.h>`.

Для виконання першого комп'ютерного практикуму потрібні будуть функції, представлені у табличці 7.1. Більшість з них повертають `double` і мають аргументи типу `double`.

Таблиця 7.1 Основні математичні функції

Функція	Призначення
`exp(x)`	Експонента $x - e^x$
`log(x)`	Натуральний логарифм числа $x - ln\ x,\ x>0$
`log10(x)`	Десятковий логарифм числа $x - log\ x,\ x>0$
`pow(x,y)`	Піднесення числа x до степеня $y - x^y$
`fabs(x)`	Модуль числа $x - \|x\|$
`sqrt(x)`	Корінь квадратний числа $x - x2$
`ceil(x)`	Округлення числа x, найменше ціле $\geq x$, *double*
`floor(x)`	Округлення числа x, найбільше ціле $\leq x$, *double*
`sin(x)`	Синус числа $x - sin(x)$, значення в радіанах
`cos(x)`	Косинус числа $x - cos(x)$, значення в радіанах
`tan(x)`	Тангенс числа $x - tg(x)$ значення в радіанах
`asin(x)`	Арксинус числа $x - arcsin(x)$ діапазон $[-\pi/2, \pi/2],\ x[-1,1]$
`acos(x)`	Арккосинус числа $x - arccos(x)$, діапазон $[0, \pi],\ x[-1,1]$
`atan(x)`	Арктангенс числа $x - arctg(x)$, діапазон $[-\pi/2, \pi/2]$
`fmod(x,y)`	Остача від ділення x на y, результат *double*
`abs(x)`	Повертає *int* і аргумент *int*

Звернути увагу, що функції **`ceil(x)`** та **`floor(x)`**, призначені для округлення числа **`x`**, повертають дійсні значення, що окремо існують функції **`fabs(x)`** та **`abs(x)`** для різних типів аргументів.

Деякі зауваження та типові помилки у перших обчислювальних програмах.

Для обчислення експоненти є спеціальна функція. Якщо просто потрібне значення числа **`e`**, можна скористатися функцією **`exp(1)`.**

Для обчислення числа **`π`** у стандарті мови немає ніяких констант, тому можна просто записати свою константу або скористатись формулою, наприклад такою

```
long double pi=acos((long double)-1);
```

або

```
pi=4.*atan(1.);
```

Для обчислення x^y можна скористатися функцією **pow(x,y).** Обидва аргументи і результат мають тип **double.** Якщо степінь **y** – ціле число і, найчастіше це квадрат або куб, то краще просто перемножити число **x** на себе.

7.4 Використання макропідстановок

Використання директиви препроцесору **#define** дозволяє вводити у текст програми константи і макровизначення. Загальний синтаксис оголошення наступний:

```
#define ідентифікатор підстановка
```

Тут **підстановка** визначає той рядок, який процесор має підставити замість рядка **ідентифікатор**, якщо зустріне такий у тексті програми.

Якщо це *макроконстанта*, то її тип буде визначений автоматично за формою. Наприклад:

```
#define A 3
#define PI 3.14159265
```

Тут **3** буде визначено як **int**, **3.14159265** – як **double.** Для кращого розуміння тексту програм, **ідентифікатор** у директиві краще виділяти великими буквами.

Якщо у програмі декілька разів потрібно виконати якусь послідовність дій можна скористатись *макрокомандою* або *функціональним макровизначенням.*

Синтаксис макрокоманди з параметрами наступний:

```
#define ідентифікатор(сп.параметрів) підстановка
```

Скрізь у тексті програми, де препроцесор зустріне послідовність **ідентифікатор(сп.параметрів),** замінить його на рядок **підстановка**, а кожний формальний параметр під час підстановки замінить на фактичний. Наприклад, якщо потрібно підводити різні значення до квадрату, можна використати таку макрокоманду:

```
#define SQR(x) ((x)*(x))
```

Приклад програми обчислення виразу

$$P=\ln(a^2+(b+1)^2)+e^a$$

```
#include <stdio.h>
#include <math.h>
#define SQR(x) ((x)*(x)
 int _tmain(int argc, _TCHAR* argv[]){
 double a, b;
       printf("enter a and b\n");
       scanf("%lf %lf", &a, &b);
         double res=log(SQR(a)+SQR(b+1))+exp(a);
       printf("result=%f\n",res);
 return 0;
 }
```

Аналогічно можна замінити макрокомандами будь-який вираз, який часто застосовують у даній програмі, для зручності сприйняття тексту, наприклад:

```
#define ABS(x) ((x)<0?-(x):(x))
#define MAX(x,y) ((x)<(y)?(x):(y))
```

Використання функцій і макрокоманд має принципові відміни, оскільки макрокоманди використовує препроцесор , який не виконує перевірки узгодженості типів, тому до їх використання потрібно ставитися обережно.

Словник. Лінійна програма, оператор, оператор-вираз, присвоювання, функції стандартної бібліотеки, директива **`#include`,** макрокоманда.

Завдання на СРС: Директиви препроцесора. Директива **`#define`** [3, с.93, 4,с.223]

Контрольні запитання

1. Що таке керуючий оператор і керуюча конструкція?
2. Що таке складений оператор?
3. Що таке вираз?
4. Які функції стають доступними при підключенні файлу **`<math.h>`** і як їх використовувати?

Лекція №8

Тема лекції: Використання керуючих конструкцій мови С. Реалізація розгалужень.

План лекції

1. Операції порівняння та умовні операції
2. Умовний оператор **`if`**
3. Оператор вибору **`switch`**

Зміст лекції

Для реалізації розгалужень в мові С передбачені дві конструкції – умовний оператор **`if`** і оператор вибору **`switch`.**

Умови в цих операторах задають з використанням так званих операцій порівняння та логічних операцій, які входять до синтаксису мови.

8.1 Операції порівняння та логічні операції

Операції порівняння – це бінарні операції, в яких значення двох змінних порівнюються один з одним.

```
>     більше ніж
>=    більше або дорівнює
<     менше ніж
<=    менше або дорівнює
==    дорівнює
!=    не дорівнює
```

Використання *логічних операцій* дає можливість реалізувати операції формальної логіки засобами мови програмування С.

! унарна операція логічне НІ – заперечення , таблиця істинності:

x	!x
0	1
1	0

Бінарні операції

&& логічне І – кон’юнкція (∧)

|| логічне АБО – диз’юнкція (∨)

Таблиця істинності виглядає так:

x	y	x&&y	x||y
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

8.2 Умовний оператор `if`

Оператор умовного переходу **if** використовують для розгалуження процесу обчислень на два напрямки. Синтаксис оператора:

```
if (вираз) оператор_1;
else
  оператор_2;,
```

де **`вираз`** — це вираз, який має логічне значення (**`true`** — «істинний» або **`false`** — «хибний»). Виконання оператору **`if`** реалізоване так: спочатку компілятор обчислить **`вираз`** і, якщо його значення не дорівнює нулю («істина»), виконується **`оператор_1,`** у протилежному випадку — **`оператор_2,`** після чого управління буде передано наступному за умовним оператору, наприклад:

```
if ( i < j ) i++;
else{
      j = i-3; //оператор_2 - складений
      i++;
};
printf(" i=%d, j=%d\n", i, j);
```

У загальному випадку **`вираз`** може не використовувати операцій порівняння, наприклад:

```
if(n++)m++;else k++;
```

Оператор **`if`** може мати скорочену форму запису, тоді у записі оператора друга частина, тобто **`else,`** відсутня. Синтаксис такого оператора наступний:

```
if (вираз) оператор_1;
```

Для такого оператора, якщо **вираз** приймає значення «неправда», що відповідає значенню 0, управління одразу буде передане на наступний після **if** оператор програми, наприклад :

```
n=0;
if (n) n++;                    //оператор_2 відсутній
printf(" n=%d,\n", n);  // n++ не виконався, n==0
```

Якщо у будь-якій гілці розгалуження необхідно виконати декілька операторів, їх слід розташовувати у блоці, як у першому прикладі.

Синтаксис С припускає, що у випадку застосування вкладених умовних операторів, при цьому кожне **else** відповідає найближчому до нього попередньому **if.** Наприклад, розглянуті фрагменти тотожні:

```
if (a > b){                          if (a > b)
  if (a > c) max = a;                    if (a > c) max = a;
    else max = c;                          else max = c;
}
```

У деяких випадках замість оператора **if** зручніше використати тернарний оператор **"?"**. Тобто оператор

```
if  (вираз)  {оператор_1} [else {оператор_2}];
```

можна записати так:

```
вираз   ? оператор_1 : оператор_2;
```

Приклад застосування тернарного оператора замість оператора **if**

```
x=10;                                x=10;
if (x<9) y=100;                      y = (x<9) ? 100 : 200;
else y=200;
```

У наведеному прикладі результати роботи обох фрагментів будуть тотожними:

Приклад . Розглянемо як приклад розв'язок добре відомого рівняння

$$ax^2 + bx + c = 0$$

Математична модель:

1) Якщо *a=0, b=0 і c=0* – безліч розв’язків.
2) Якщо *a=0, b=0, c≠0* – немає розв’язку.
3) Якщо *a=0 b≠0* – лінійне рівняння $bx + c = 0$ і один розв’язок $x = -\frac{c}{b}$.
4) Якщо *a≠0* – квадратне рівняння і якщо детермінант $D = b^2 + 4ac < 0$, то немає розв’язку серед дійсних чисел, якщо *D≥0* – два корені

$$x_1 = \frac{-b+\sqrt{D}}{2a}; x_2 = \frac{-b-\sqrt{D}}{2a};$$

Алгоритм розв'язку такої задачі можна побудувати двома шляхами – з використанням так званої квазілінійної моделі, яка практично відповідає побудованій математичній моделі, або з використанням вкладених умов.

Перший алгоритм передбачає використання у програмі складених логічних виразів, наприклад

```
printf("enter a,b,c ");
scanf("%lf%lf%lf", &a, &b, &c);
if((a==0)&&(b==0)&&(c==0))
       printf("set of solutions");
if((a==0)&&(b==0)&&(c!=0))
       printf("no solutions");
if((a==0)&&(b!=0))
       printf("linear equation x=%f\n", -c/b);
if(a!=0) {
         d=b*b+4*a*c;
         if(d<0)
              printf("no valid solutions");
              else {
                   x1=(-b+sqrt(d))/(2*a);
                   x2=(-b-sqrt(d))/(2*a);
                   printf("roots x1=%f, x2=%f\n", x1, x2);
              }
    }
```

Якщо значення дійсного числа не є щойно уведеним з клавіатури, а отримане в результаті обчислень, які завжди накопичують похибку, варто перевірку організувати з врахуванням точності. Для цього або уводять з клавіатури, або задають як константу (частіше – макроконстанту) деяке число ε.

Наприклад

$$y = \begin{cases} \dfrac{25.5}{x-1}, x \neq 1 \\ 1, інакше \end{cases}$$

```
#define EPSILON 0.0001
. . .
if(fabs(x-1)>epsilon)
          y=25.5/(x-1);
     else     y=1.0;
```

8.3 Оператор switch

Довгі ланцюжки операторів **if-else** важкі для сприйняття, тому, якщо варіант вибору шляху подальшого обчислення залежить від цілих значень, використовують оператор *вибору* (*перемикач*). Синтаксис:

```
switch (вираз_цілого_типу)
{
```

```
    case константний вираз 1: [оператор1; ] [break;]
    case константний вираз 2: [оператор2; ] [break;]
    …
    case константний вираз n : [оператор n; ] [break;]
    [default:  оператор n+1]     // аналог else в if
}
```

Семантика оператора: після обчислення **виразу_цілого_типу** його значення почергово порівнюється з кожним з **константних_виразів** і, коли відбудеться співпадіння, наприклад, з **виразом i**, управління передається на **оператор i**. Якщо після оператора стоїть оператор **break**, то після виконання **оператора i** управління передається на наступний за **switch** оператор. Якщо жодного співпадіння не відбулося, управління передається на мітку **default** за її наявності або просто жодна з гілочок оператора виконана не буде і виконання програми продовжиться з наступного за **switch** оператора.

Оператор **break**; відноситься до операторів передачі управління і приводить до переривання виконання оператора, у якому він використовується. Управління передається наступному оператору. Використовується в **switch** і в циклах.

Приклад. Виведення назв перших трьох чисельників англійською мовою

```
 switch (number)
 {
    case 1:    printf(" % d - one \n", number);break;
    case 2:    printf(" % d - two \n", number);break;
    case 3:    printf(" % d - three\n", number);break;
    default: printf ("good bye \n");
}
```

Як правило, оператор вибору використовують саме у такому форматі, з **break.** Але оскільки з **case** завжди пов'язане тільки одне значення, а інколи потрібно виконати однакові дії для двох або більше міток, тоді **break** не використовують. Якщо оператор **break** відсутній, то управління передається на наступний **case** і так до закінчення всього оператора **switch** включно з **default**, або до появи першого оператора **break** . Наприклад, програма переведення оцінки із 12-бальної у 5-бальну.

```
int main(){
   int n;
   printf("enter your mark\n");
   scanf("%d", &n);
```

```
   switch (n){
       case 0: case 1: case 3: printf("bad\n");   break;
       case 4:
       case 5: case 6: printf("satisfactory\n");  break;
       case 7:
       case 8: case 9: printf("good\n");break;
       case 10:
       case 11: case 12: printf("excellent\n");   break;
       default: printf ("incorrect mark\n");
     }
   return 0;
}
```

Дві різні мітки не можуть мати однакові значення. Вираз умови в інструкції **switch** може бути як завгодно складним, у тому числі включати виклики функцій. Оператори вибору можуть бути вкладеними один у другий і в інші конструкції мови.

Словник. Розгалуження, умовний оператор, тернарний оператор, операція порівняння, табличка істинності, логічний оператор, складна умова, блок, перемикач, оператор вибору, оператор виходу із оператора, вкладеність.

Завдання на СРС: Директиви умовної компіляції [3, с.93, 8,с.223, 4, с.245]

Контрольні запитання

1. Якими операторами може бути реалізоване розгалуження у програмі?
2. Опишіть оператор вибору **switch**.
3. Навіщо потрібен оператор **break** в конструкції **switch**?
4. Чи можна об’єднувати розділи **case**?
5. Чи можна вкладати умовні оператори один в другий?

Лекція №9

Тема лекції: Використання керуючих конструкцій мови С. Оператори циклу

План лекції

1. Оператори циклу
2. Цикл з передумовою **`while`**.
3. Цикл з постумовою **`do while`**

Зміст лекції

9.1 Оператори циклу

Оператори циклу використовують для здійснення багаторазового повторення деякої послідовності дій.

Кожен цикл складається з *умови* та *тіла* циклу, що виконується декілька разів. Один прохід циклу називається *ітерацією*. У мові С існують три циклічні конструкції: **`while, do while, for.`**

9.2 Цикл з передумовою **`while`**

Відповідно до теорії будь-яка структурна програма може бути реалізована за допомогою лінійних блоків, умовних блоків і циклів з передумовою. Тобто цикл з передумовою – основна циклічна конструкція, яка може замінити будь-яку. Але для зручності ввели також і інші.

Цикл називається циклом з передумовою тому, що умова в ньому перевіряється до початку наступної ітерації циклу

Синтаксис циклу з передумовою:

```
while (вираз) оператор;
```

Тіло циклу представлено **`оператором`**, який може бути простим або складеним. **`Складеним`** називається послідовність операторів, розділених знаком «**`;`**», які сформовані у **`блок`**, обмежений фігурними дужками.

Порядок виконання оператора наступний.

Спочатку буде перевірена умова входу у цикл. Якщо умова істинна, тобто значення **`виразу`** не дорівнює нулю, буде виконаний **`оператор`**, після чого управління знову буде передане на початок циклу – перевірку умови входу в цикл.

Так цикл буде повторюватись поки умова входу не стане хибна, тобто значення **виразу** стане рівним нулю, тоді цикл закінчиться і управління одразу буде передане на наступний за ним оператор.

Приклад використання:

```
int i=0, sum=0;
while (i < n)
  sum += ++i * i;//простий оператор
```

Оператор **while** зручно застосовувати у випадках, коли кількість ітерацій заздалегідь невідома. Наприклад, потрібно обчислити суму ряду $\sum_{k=1}^{\infty} \frac{1}{k^2}$, поки черговий член ряду не стане меншим за деяке наперед задане число ε.

```
#define SQ1(x) 1./((x)*(x))
int main()
{
  double s=0, eps;
  int k=1;
  printf("enter epsilon\n");
  scanf("%lf", &eps);
  while(SQ1(k)>eps)
  {
       s=s+SQ1(k);
       k++;
  }
  printf("result=%f \n", s);
  return 0;
}
```

Якщо значення **виразу** одразу дорівнюватиме нулю, цикл з передумовою не виконається жодного разу, управління буде передане до наступного за цим циклом оператора. Наприклад, якщо у розглянутому вище прикладі помилково увести **ε>1**.

9.3 Цикл з постумовою do while

Циклічну конструкцію з постумовою застосовують у випадках, коли тіло циклу потрібно виконати хоча б один раз. Тіло циклу представлено простим або складеним **оператором**, умова входу в цикл перевіряється **після** його виконання. Синтаксис:

```
do оператор while (вираз);
```

Так само, як і в циклі з передумовою, умова виконання циклу – ненульове значення **виразу**.

Синтаксисом циклу **do while** не передбачені обов'язкові фігурні дужки для випадку з одним оператором, але їх використання є бажаним для забезпечення більшої читабельності програми.

Типові випадки використання циклів з постумовою – організація меню, перевірка коректності уведених даних, наближені обчислення за ітераційною формулою з заданою точністю, тощо.

Приклад використання для забезпечення коректності введення інформації:

```
int numb;
do{
      printf("enter the number of students \n");
      scanf("%d",&numb);
 }while(numb<0);
 printf("ok, number of students=%d\n", numb);
```

Як і для циклу з передумовою, робота циклу з постумовою буде закінчена по нульовому значенню виразу.

Приклад використання для обчислення суми ряду з заданою точністю. Дано дійсні числа x, ε $\left(x \neq 0, 0 < \varepsilon < 1\right)$. Обчислити з точністю до ε та дослідити збігання послідовності для обраного значення x з виведенням кожного p-го доданку та відповідного значення суми:

$$\sum_{k=0}^{\infty} \frac{(-1)^{k+1}(x/2)^{2k}}{2(k+1)!} .$$

Для розв'язку даної задачі варто спочатку побудувати математичну модель. $\frac{a_{k+1}}{a_k} = \frac{(-1)\left(\frac{x}{2}\right)^2}{k+2}$, тоді $a_{k+1} = \frac{(-1)\left(\frac{x}{2}\right)^2}{k+2} a_k$ і обчислення проводитимуться поки черговий a_k не стане менше деякого наперед заданого ε. Потрібно увести x, p, ε.

```
    int main(){
      double s=0, a=1., x, eps;
      int k=0, p;
printf("enter x and p\n");
scanf("%lf%d", &x, &p);
      do
      {
           printf("enter 0<epsilon<1\n");
           scanf("%lf", &eps);
      } while(fabs(eps)>1);
      do
      {//формування наступного члена ряду через попередній
           a=-a*x*x/(4*(k+2));
           s=s+a;
```

```
            if(k%p); //чи це p-тий член ряду?
                 else printf("a(%d)=%f, s=%f\n", k, ak, S);
            k++;
} while(fabs(a)>eps);
      printf("s=%f \n", s);
      return 0;
    }
```

Можливий сеанс роботи програми

```
enter x and p
3 2
enter 0<epsilon<1
0.0001
a(0)=-1.125000, s=-1.125000
a(2)=-0.474609, s=-0.755859
a(4)=-0.080090, s=-0.622375
a(6)=-0.007240, s=-0.603872
a(8)=-0.000407, s=-0.602470
s=-0.602386
```

На цьому прикладі видно, що для **x** =3 і **eps**=0.0001 ряд сходиться. У прикладі цикл з постумовою був використаний двічі для ілюстрації варіантів його типового застосування.

Словник. Цикл, умова продовження циклу, тіло циклу, ітерація циклу, передумова, постумова, безкінечний цикл.

Контрольні запитання

1. Які різновиди визначення умови входу в цикл в С ви знаєте?
2. Що таке тіло циклу? Ітерація циклу?
3. Які принципові відміни між циклами з перед- і постумовою?
4. У якому випадку цикл не буде виконано жодного разу?
5. У якому випадку цикл буде виконано хоч раз за будь-яких умов?

Лекція №10

Тема лекції: Використання керуючих конструкцій мови С. Цикл з параметрами

План лекції

1. Цикл **for**
2. Порядок виконання оператору **for**
3. Безкінечні цикли
4. Вкладені цикли
5. Порівняльна характеристика операторів циклу в С

Зміст лекції

10.1 Цикл for

Синтаксис циклу **for** наступний:

```
for([ініціалізація];[вираз];[модифікація])[оператор];
```

У секції **ініціалізації** виконують оголошення параметрів циклу у вигляді списку, з використанням операції «кома». Цих параметрів може бути декілька , обмежень на типи немає. В С++ та пізніших специфікаціях С вони можуть бути тут же і ініціалізовані, якщо їх використання у програмі ніде крім даного циклу не заплановане.

Вираз – це умова виконання циклу. У цій секції може бути лише один вираз. Це може бути складеною умовою, але операція кома тут не допустима.

Секція **модифікацій** виконується після кожної ітерації циклу і слугує для зміни значень параметрів. Виразів модифікації може бути декілька, тобто можна використовувати операцію кома.

Оператор визначає тіло циклу. Він може бути як простий, так і складений, тоді стають обов'язковими фігурні дужки.

Приклад циклу:

```
int n=10, y, k;
for (k = 1; k <= n; k++)
     printf("k=%d, k^2=%d", k, k*k);
```

У наведеному прикладі на екран будуть виведені цілі від 1 до 10 і їх квадрати.

10.2 Порядок виконання оператору `for`

Спочатку буде виконаний етап ініціалізації циклу, вираз за виразом, зліва направо, якщо секція ініціалізації не порожня. Ця секція виконується один раз, на самому початку роботи циклу. Приклад циклу зі списком у секції ініціалізації:

```
int n, y, k;
for (k = 0, n = 20; k <= n; k++, n--)
     y = k * n;
```

Потім буде обчислено значення **виразу** – виразу умови продовження циклу.

Якщо значення цього виразу відмінно від нуля (тобто істинно), буде виконаний **оператор** циклу. Це етап виконання тіла циклу. Якщо значення виразу умови продовження циклу хибне, виконання циклу переривається і управління передається на наступний після **for** оператор.

Після цього будуть обчислені значення виразів, розташованих у секції модифікації. Це етап обчислення параметрів циклу для здійснення наступної ітерації циклу.

На останніх двох етапах значення раніше визначених змінних можуть бути змінені, тому наступна ітерація циклу буде розпочата з етапу визначення умови продовження циклу.

Цей цикл є циклом з передумовою, оскільки умова входу в цикл перевіряється перед початком чергової ітерації. Відповідно, вираз, що визначає умову може одразу дорівнювати нулю і тоді вхід до циклу не відбудеться взагалі, управління буде передане наступному за циклом оператору.

Кожна з секцій може бути відсутня, як і всі одночасно. Але знаки `;` у дужках після **for** є обов'язковими. Наприклад

```
int n=10, y=0, k=0;
for (;k <= n; k++, n--) y += k * n;
```

У наведеному прикладі відсутня перша секція.

10.3 Безкінечні цикли.

Якщо умова входу у цикл побудована некоректно, цикл може ніколи не закінчитися, наприклад:

```
for (k = 0, n = 5; k <= n; k++, n++)
                                        y = k * n;
```

Інколи безкінечний цикл – це не помилка, просто потрібно організувати цикл, який ніколи не повинен закінчитися, окрім випадку з певною нестандартною ситуацією.

Наступний цикл, що виводить синуси уведених кутів, працюватиме до тих пір, поки не буде уведений нечисловий символ.

```
double angle;
 for(;;)
 {
      printf("enter angle \n");
      if(scanf("%lf", &angle))
               printf("sin(%f)=%f\n", angle, sin(angle));
      else break;
 }
```

Тут **scanf** поверне 0, якщо буде уведене не число.

10.4 Вкладені цикли

Цикли можуть бути вкладеними, якщо виникає необхідність багаторазового виконання певної дії, яка сама по собі також представляє циклічну конструкцію.

Глибина вкладеності мовою не регламентована. Програміст повинен сам стежити, щоб границі зовнішнього і внутрішнього циклів не перетиналися. Вкладеними можуть бути цикли різних типів.

Наприклад, фрагмент програми з реалізацією імітації годинника з виведенням поточного значення годин, хвилин і секунд.

```
      int h=0, m, s;
printf("hours, minutes, seconds\n");
      while(h<24){
         for(m=0; m<60; m++)
              for(s=0; s<60; s++)
                  printf("%5i, %6i, %6i\n", h, m, s);
           h++;
      }
```

Результати будуть виведені у стовпчик.

10.5 Порівняльна характеристика операторів циклу

Оператори циклу в С є взаємозамінні, відміни у використанні просто надають певні зручності для користувача. Цикли **while** і **for** – цикли з передумовою і тому може виникнути ситуація, коли такий цикл не буде

виконаний жодного разу. Цикл **`do while`** – цикл з постумовою і тому тіло циклу обов’язково буде виконано хоча б один раз.

Словник. Параметр циклу, ініціалізація, модифікація, умова продовження, оператор «кома».

Завдання на СРС: Оголошення і використання макросів [2,с74, 3,с.95, 6,с. 246, 4,с.226]

Контрольні запитання

1. Опишіть синтаксис і призначення кожної секції оператора **`for`**.
2. Опишіть механізм дії оператора **`for`**.
3. Що таке лічильник циклу?
4. Скільки і якого типу може бути лічильників у циклі **`for`** в С?
5. Параметри яких типів можна використовувати у циклі for?
6. Чи допустимо використання оператора кома при визначенні умови виконання циклу?
7. Які обмеження на глибину вкладеності циклів ви знаєте і з чим вони пов’язані?
8. Що таке «безкінечний цикл»?У яких випадках можна використовувати безкінечні цикли?
9. Про що потрібно пам'ятати, щоб не організувати безкінечний цикл?
10. Які різновиди циклів С ви знаєте і чим вони відрізняються?

Лекція №11

Тема лекції: Використання керуючих конструкцій мови С. Оператори керування обчислювальним процесом.

План лекції

1. Оператор **break**
2. Оператор **continue**
3. Оператор **return**
4. Оператор **goto**
5. Функція **exit()**

Зміст лекції

У стандарті мови визначені наступні оператори керування: **break**, **continue**, **goto**, **return**. До цього переліку можна віднести і функцію **exit()** (заголовковий файл <**stdlib.h**>**)**

11.1 Оператор break

Оператор **break** використовують в умовних операторах і усіх типах циклів. Виклик **break** закінчує виконання поточного оператора у даній позиції без будь-яких перевірок і передає управління наступному оператору. Якщо **break** належить внутрішньому циклу або вкладеному умовному оператору, управління буде передане до найближчого зовнішнього циклу.

У наведеному далі прикладі вихід із безкінечного циклу реалізований шляхом передачі управління на наступний оператор.

```
while(1){
      scanf("%lf",&eps);
      if(eps<1&&eps>0) break;
}
```

Наступний приклад демонструє ситуацію передачі управління зовнішньому циклу:

```
for (int i = 1; i < 5; i++){
          printf("\n %d :", i);
          for (int j = 1; j < 5; j++){
               if (i+j==5) break;
               printf("%d ", j);
           }
      }
  Результат:
   1 :1 2 3
```

```
2 :1 2
3 :1
4 :
```

У наведеному прикладі видно, що внутрішній цикл повністю завершений оператором **break** і наступний старт цього циклу відбудеться з початку і тільки з наступної ітерації зовнішнього циклу.

Оператор **break** найчастіше використовують в операторі вибору. Приклади були розглянуті раніше.

11.2 Оператор continue

Оператор **continue** перериває тільки поточну ітерацію циклу. Цей оператор використовують тільки у циклах **do while**, **for** або **while**.

Наступний приклад демонструє відміни у результатах використання операторів **break** і **continue**. Це модифікація попереднього прикладу для **continue**:

```
for (int i = 1; i < 5; i++)
   for (int j = 1; j < 5; j++){
        if (i+j==5)   continue;
       printf("%d ", j);
    }
```

результат буде таким:

```
1 :1 2 3
2 :1 2 4
3 :1 3 4
4 :2 3 4
```

Після оператора **continue** усі інші оператори поточної ітерації не виконуються, а наступна ітерація циклу визначена так:

– для **do** або **while** починається з повторного обчислення керуючого виразу;

– для **for** – з виконання секція модифікації, а потім повторного виконання перевірки входу в цикл, залежно від результату якої або починається наступна ітерація або цикл завершується.

11.3 Оператор return

Оператор **return** завершує виконання функції і повертає управління у точку виклику даної функції, тобто до тієї, з якої її було викликано, або в операційну систему при передачі управління із функції **main**. Якщо після

ключового слова **return** міститься деякий вираз, то це результат роботи функції. Розгляд роботи функцій передбачене у лекції 21.

11.4 Оператор goto

Оператор **goto identifier** безумовно передає управління в інструкцію з міткою **identifier**. Синтаксис:

```
goto identifier;
. . .
identifier: оператор;
```

Ідентифікатор мітки **identifier** – звичайний ідентифікатор і повинен бути унікальним не тільки серед інших міток, але й серед інших ідентифікаторів даної функції.

Оператор з ідентифікатором **identifier** може знаходитися у довільному місці відносно положення оператора **goto**, але обов'язково у поточній функції.

Мітка **identifier** має значення тільки при переході для **goto,** в інших випадках буде проігнорована.

Повторно оголошувати мітку не можна.

```
for ( i = 0; i < 10; i++ ){
     printf( "Outer loop executing. i = %d\n", i );
     for ( j = 0; j < 2; j++ ){
        printf( " Inner loop executing. j = %d\n", j );
          if ( i == 3 )
              goto stop;
      }
 }
    printf( "Loop exited. i = %d\n", i );//not print:
    stop:  printf( "Jumped to stop. i = %d\n", i );
```

результат:

```
Outer loop executing. i = 0
 Inner loop executing. j = 0
 Inner loop executing. j = 1
Outer loop executing. i = 1
 Inner loop executing. j = 0
 Inner loop executing. j = 1
Outer loop executing. i = 2
 Inner loop executing. j = 0
 Inner loop executing. j = 1
Outer loop executing. i = 3
 Inner loop executing. j = 0
Jumped to stop. i = 3
```

На відміну від **break**, яка дає можливість вийти з одного рівня, оператор **goto** дозволяє вийти з любого рівня вкладеності, як у даному прикладі.

11.5 Функція `exit()`

Функція **`exit()`** із **`<stdlib.h>`** перериває виконання програми і передає управління одразу до операційної системи (ОС), в якій ця програма була запущена.

Цю функцію використовують для керування пристроями у більшості програм. Синтаксис:

```
void exit (int статус);
```

До ОС функція *повертає* значення статусу: коректне завершення – 0, інші значення аргументу використовують для ідентифікації певних помилок .

Частіше всього **`exit()`** використовують, коли умови виконання програми незадовільні. Наприклад комп'ютерна гра може вимагати спеціальний графічний адаптер. Функція **`main()`** даної гри може виглядати наступним чином:

```
#include <stdlib.h>
int main(){
 if (!special()) exit(1);
 play ();
return 0;
}
```

тут **`special()`** – це **визначена** користувачем функція, яка повертає істину, якщо необхідний адаптер присутній. Якщо ж відсутній – функція повертає 0 і програма завершує роботу.

Словник. Безумовний перехід, рівень вкладеності, аварійний вихід, переривання, повернення із функції.

Завдання на СРС: Особливості використання функцій **`exit()`** і **`abort()`**, визначених у заголовковому файлі **`<stdlib.h>`** [3,с.103, 2,с.41]

Контрольні запитання

1. Назвіть чотири оператори безумовного переходу.
2. Для чого призначений оператор **`return`**? оператор **`break`**?
3. Опишіть відмінність оператора **`continue`** від оператора **`break`**.
4. Опишіть механізм дії оператора **`goto`**.
5. Опишіть функції **`exit()`** та **`abort()`**. Як вони функціонують?

Лекція №12

Тема лекції: Оголошення вказівників. Звертання до даних через вказівники

План лекції

1. Визначення вказівника
2. Вказівники на об’єкти
3. Адресні операції
4. Ініціалізація вказівників
5. Перетворення типу для вказівників

Зміст лекції

12.1 Визначення вказівника

Вказівник – іменований об'єкт, призначений для зберігання адреси області пам'яті (об'єкта, неіменованої області оперативної пам'яті або точки входу у функцію).

Вказівник не є самостійним типом, він завжди пов'язаний з якимось іншим типом. Вказівники діляться на дві категорії:

- вказівники на об’єкти;
- вказівники на функції.

Ці категорії вказівників відрізняються один від одного властивостями і правилами маніпулювання.

12.2 Вказівники на об’єкти

Кожний вказівник має відповідний тип. Загальний синтаксис оголошення вказівника на об'єкт:

```
тип * ідентифікатор_змінної
```

Тип задає тип об'єкта, адресу якого міститиме оголошувана змінна і може бути базового, у тому числі і типу **`void`**, структурованого типу (перерахування, структура, об'єднання).

Реально вказівник на **`void`** ні на що не вказує, але має здатність вказувати на область будь-якого розміру після його типізації яким-небудь об'єктом.

Ідентифікатор визначає ім'я оголошуваної змінної типу вказівник або конструкцію, яка організовує безпосередній доступ до пам'яті. Ідентифікатору обов'язково повинна передувати зірочка **`(*)`**. Тому у списку описування

змінних зірочка повинна передувати кожному ідентифікатору вказівника. Наприклад:

```
int *p, a, *q;   // тут p і q – вказівники
```

Об'єм пам'яті, який виділяється для зберігання даних, визначений типом даних. Розмір пам'яті, яку займає вказівник, визначений моделлю пам'яті і не залежить від типу змінної, на яку цей вказівник вказує. Це може бути 2, 4 або 8 байтів. Для його визначення простіше всього скористатися операцією **sizeof(ідентифікатор_вказівника).**

У мові С визначено декілька операцій для роботи з вказівниками. Це адресні операції, арифметичні і порівняння.

12.3 Адресні операції

Визначені дві унарні адресні операції: взяття адреси та розадресації.

- Адресна операція *взяття адреси* **&** визначає адресу змінної у пам'яті.

Синтаксис:

```
ідентифікатор_вказівника = & ідентифікатор_змінної
```

Наприклад:

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

В результаті застосування у змінну-вказівник **p** буде занесена адреса змінної **a.** Фактично змінна **p** міститиме адресу молодшого байту ділянки пам'яті, яку займає **a.** Тип змінної **a** повинен відповідати базовому типу вказівника **p.**

Операцію **&** не можна застосовувати до таких об'єктів:

1) константного літерала будь-якого типу,
2) виразу незалежно від типу результату,
3) змінної з класом пам'яті **register**.

- Адресна операція отримання значення величини, що знаходиться за наданою адресою – операція *розадресації* (або *розіменування*). Ця операція фактично зворотна до операції **&**. І так само, як і для операції адресації, типи змінної і вказівника, що на неї вказує, повинні бути узгоджені.

Синтаксис:

```
Ідентифікатор_змінної = * ідентифікатор_вказівника
```

Наприклад: `int b =*p;`

Семантика: у змінну **b** записати значення, на яке посилається вказівник **p**.

Приклад використання:

```
int *p, a=100, *q, b;
p = & a;
b = * p;
printf("a=%d, b=%d\n", a, b);
// результат: a=100, b=100
```

Якщо для деякої змінної визначений і ідентифікатор і вказівник на неї, звертатися до неї можна з використанням і того і другого, результат буде однаковим. У наведеному раніше прикладі звертання **a** і ***p** є тотожними:

```
*p = 200;
printf("a=%d,*p=%d\n",a,*p);
//результат: a=200, *p=200
```

Вказівник може бути константою або змінною, а також вказувати на константу або змінну.

Прикладами константних вказівників можуть бути ідентифікатори масивів і рядкових літералів.

Приклади оголошень:

```
int i;// ціла змінна
const int ci=1; // ціла  константа
int *pi; //вказівник  на цілу змінну
const int *pci; // вказівник  на цілу константу
int *const cpi; // вказівник-константа на цілу змінну
const int *const cpc;//вказівник-константа на цілу константу
```

12.4 Ініціалізація вказівників.

Неініціалізований вказівник – джерело найбільшої кількості помилок, які важко діагностувати. Способи ініціалізації вказівників наступні.

1. Присвоїти вказівнику адресу існуючого об’єкту

– з використанням адресної операції, наприклад
```
int a=5;
int *p=&a; // або int *p(&a)
```
– з використанням значення іншого ініціалізованого вказівника, наприклад
```
int *q = p;
```
– з використанням ідентифікатора масиву або функції, наприклад
```
int arr[5], *q = arr;
```

2. Присвоїти вказівнику адреси області пам’яті у явному вигляді, наприклад:

```
char *cp=(char*)0x B800 0000;
```

3. Присвоїти вказівнику порожнє значення, наприклад:

```
int *N=NULL;
```

Такий вказівник використовують для перевірки, чи посилається він на який-небудь об’єкт

4. Ініціалізація з виділенням динамічної пам’яті

- В стилі С – за допомогою функції **malloc(),** прототип якої знаходиться у заголовковому файлі **<stdlib.h>.** Наприклад для виділення пам'яті під змінну типу **int,** потрібно написати:

```
int *p =( int*) malloc(sizeof (int))
```

Для контролю значень змінних-вказівників можна скористатись специфікатором формату **%p**, який виводить фізичні значення адрес у системі з основою 16. Наприклад фрагмент програми

```
double d=200, d1=300, d2=400;
double *p=&d;
printf("%p, %p, %p, %d\n", p, &d1, &d2, sizeof(p));
```

дасть результат такого вигляду

```
0016F858, 0016F848, 0016F838, 4
```

12.5 Перетворення типу для вказівників

В С дозволено присвоювання значення одного вказівника іншому, якщо вони одного типу. Наприклад:

```
int a=5;
int *p1, *p2;
p1=&a;
p2= p1;
```

Допустиме присвоювання вказівника одного типу вказівнику другого типу за умови виконання явного перетворення типу. У цьому випадку за коректність результату відповідає користувач. Наприклад

```
double x=100.1, y;
int *p;
p=(int*)&x;
y=*p;
printf("p=%p,*p=%d\n", p, *p);//p=003AFBBC, *p=1717986918
printf(" x=%f\n", y); // x=1717986918.000000
```

У розглянутому прикладі перетворення виконано коректно, а результат отримано некоректний, тому що тип результату при роботі з вказівником визначається типом вказівника.

В С допустиме присвоювання вказівника **void*** вказівнику любого іншого типу і навпаки напряму без явного перетворення типу.

Вказівники **void*** називають *нетипізованими* вказівниками і вони сумісні з вказівниками будь-якого іншого типу.

Наприклад:

```
double z=200.1, *q;
int i=5, *p ;
void * r;
q=&z;
r=q;
p=r;
```

Нетипізовані вказівники застосовують для передачі у функцію параметрів, тип яких може бути різним при різних викликах даної функції.

В С++ вказівнику **void*** допустиме присвоювання вказівника іншого типу напряму, але для зворотного перетворення потрібне явне приведення типу. Наприклад:

```
double z=200.1, *q;
int i=5, *qi;
void * r;
q=&z;  r=q;
qi=(int*)r;
```

Результат не завжди коректний, але все дозволено. Таке перетворення як правило використовується під час передачі параметрів у функцію.

Дозволене також перетворення **int <--> int*,** якщо йдеться про присвоєння вказівнику значення адреси у явному вигляді.

Словник. Вказівник, адреса комірки пам'яті, розименування, константний вказівник, ініціалізація, взяття адреси, динамічна пам'ять, NULL.

Контрольні запитання

1. Як оголосити вказівник? Як ініціалізувати?
2. Що означає константа **NULL**? До вказівників якого типу її можна застосовувати? З якою метою її можна використовувати?
3. Що таке розіменування вказівника?

Лекція №13

Тема лекції: Адресна арифметика

План лекції

1. Масиви і вказівники
2. Операції порівняння вказівників
3. Інкремент та декремент для вказівників
4. Додавання та віднімання цілих
5. Віднімання двох однотипних вказівників

Зміст лекції

13.1 Масиви і вказівники

Ім'я масиву – константний вказівник на його перший елемент. Тобто якщо масив оголошено

```
int month[12],
```

то ідентифікатор **`month == &month[0]`** і має тип **`int * const.`**

Тобто до елементу масиву можна звернутися як за допомогою ідентифікатора і індексу, так і за допомогою вказівника. Наприклад, можна визначити не константний вказівник так

```
int *p= month;
```

після чого звертатись і через **`month`** і через **`p`**. В обох випадках можна використовувати індекси. Наприклад, звернення **`month[2]`** тотожне **`p[2]`** і **`*(p+2).`**

Це потрібно враховувати у роботі з масивами. У мові С(С++) на роботу з індексами масиву йде більше часу, ніж на застосування операції розадресації, тому використання вказівників для доступу до елементів масиву достатньо поширене.

Для вказівників крім адресних допустимі і інші операції: порівняння, присвоювання, арифметичні.

13.2 Операції порівняння

Для вказівників визначені усі операції порівняння. Операції «дорівнює» і «не дорівнює» **`(==`** і **`!=)`** можуть бути застосовані до вказівників, що посилаються на різні, хоча і однотипні об’єкти.

Інші операції порівняння доречні, якщо стосуються одного об’єкта, наприклад, масиву. Порівняння відбувається за абсолютними значеннями.

Наприклад:

```
int mon[10], *p, *q;
```

31	28	31	30	31	30	31	31	30	31

```
p▲                    q▲
p != q; p < q;
```

Порівняння двох вказівників.

13.3 Інкремент та декремент

Застосування цих операцій до вказівника приводить до збільшення або зменшення (відповідно) абсолютного значення адреси, яку цей вказівник містить, на **sizeof(базовий_тип)** байтів. Фактично вказівник пересувається на один елемент відповідного типу вперед або назад. Як правило, ці операції застосовують для забезпечення руху по елементам масиву, адже для гарантування коректності потрібно бути впевненим, що наступний або попередній елементи того самого типу, що й поточний.

Наприклад, якщо **int *m,** і змінна **m** містить значення **0x001A3404,** тоді після застосування до неї **m++** , вона міститиме **0x001A3408** (а якби **m--,** тоді містила би **0x001A3400**).

Якщо **int *m= mon** або, що тотожно, **int *m= &mon[0]**, тоді після застосування до неї **m++** , вона вказуватиме на **mon[1].**

Наступний приклад демонструє використання операції інкременту. Потрібно обнулити усі від’ємні числа спочатку масиву. Масив гарантовано містить принаймні одне додатне число. Індекси не використовувати.

```
#define MAX 100
int main()
{
  double arr[MAX];
  int *p, n, i;
  printf("enter dimension\n");
  scanf("%d", &n);
  printf("enter %d array elements\n", n);
      for(i=0; i<n; i++)
              scanf("%lf", &arr[i]);
. . .
}
        for(p=arr; *p<0; p++) *p=0;
```

Операціїї інкремента та декремента мають вищий приоритет за операцію розадресації, тобто наприклад у виразі **int d=*p++** спочатку відбудеться присвоєння **d** значення, на яке вказує **p**, а потім він пересунеться на наступний елемент.

13.4 Додавання та віднімання цілих

Так само як і для операцій інкременту та декременту, розрахунок потрібних адрес компілятор буде виконувати не у байтах, а у об'єктах відповідних типів.

Синтаксис операції додавання цілого:

ідентифікатор_вказівника+вираз_цілого_типу

У загальному випадку абсолютне значення адреси **p** після застосування операції **p+=n,** де **p** оголошено як вказівник на деякий **базовий_тип,** збільшиться на **n*sizeof(базовий_тип).**

Або так само зменшиться для **-=**. Наприклад

```
int *p;
p=p+12;
```

вказівник пересунувся на 12 об'єктів у бік збільшення адрес і абсолютне значення адреси збільшилося на 48.

Інколи оператори додавання та віднімання цілого застовують для роботи безпосередньо з ідентифікатором масиву, якщо використовують його як вказівник. Оскільки це константа, застосовувати до нього інкремент або декремент не можна, а у виразах справа від оператора присвоювання з оператором **+ ціле** або **– ціле** допустимо. Наприклад:

```
int mon[12], *p, dim=12;
for (p= mon; p<mon+dim; p+=3)
      if(*p>0)  printf("last=%d\n",*p);
```

У розглянутому циклі додавання **mon+dim** не приводить до зміни значення константного вказівника, отже допустиме.

13.5 Віднімання двох однотипних вказівників

Цю операцію використовують для визначення кількості елементів, розташованих між ними. Зрозуміло, що вказівники, до яких застосовують цю операцію, повинні вказувати на елементи одного масиву. Наприклад:

```
int *p=arr, *q=p+n;
while((*p!=*q)&&(p<q)){p++;q--;}
if(p<q)printf("dist=%d\n", q-p);
```

Наведений приклад демонструє визначення кількості елементів масиву, розташованих між двома однаковими

Контрольні запитання

1. Які операції називають операціями адресної арифметики?
2. Як змінюється значення вказівника при застосуванні до нього операції інкременту?
3. Як можна звернутися до елементу масиву за допомогою вказівника на його нульовий елемент?
4. Як обчислити різницю значень двох вказівників?
5. За яких умов до вказівників можна застосовувати оператор присвоювання?

Розділ 4. Структурований тип масив

Лекція №14

Тема лекції: Масив як структурований тип. Одновимірні масиви

План лекції

1. Основні характеристики масиву
2. Одновимірні масиви
3. Оголошення та ініціалізація

Зміст лекції

14.1 Основні характеристики масиву

Масив – іменована послідовність областей пам'яті, що містять однотипні елементи, розташовані підряд. Кожна така область пам'яті називається *елементом* масиву. Для масивів визначений *вимір*, який дозволяє описувати одновимірні, двовимірні, багатовимірні масиви та *розмірність* (більша або дорівнює одиниці), яка задає кількість елементів у одному вимірі масиву. Загальна кількість елементів масиву дорівнює *вимір* ***** *розмірність* і називається *розміром* масиву.

14.2 Одновимірний масив

Одновимірний масив або вектор має єдиний вимір. Синтаксис оголошення одновимірного масиву наступний:

```
тип ідентифікатор[константний_вираз]
```

Тут **тип** визначає тип елементів масиву і називається базовим типом, **ідентифікатор** це назва масиву, а **константний_вираз** – обов'язкова частина оголошення – визначає його **розмір**, тобто кількість елементів масиву.

До кожного з цих елементів можна звернутися за ідентифікатором масиву та порядковим номером елементу – його **індексом**.

Оскільки елементи масиву у пам'яті розташовані підряд, розмір пам'яті, виділеної під масив, визначений так:

```
розмір*sizeof(тип).
```

Наприклад, для масиву, оголошеного як

```
double balance[100]
```

розмір пам'яті дорівнюватиме 100*8=800 байт.

Індекс визначає порядковий номер елементу масиву і може бути будь-яким цілим виразом, наприклад **`balance[25], balance[2*j+k].`**

Індексація починається з нуля, завершується значенням **`розмір-1,`** тобто **`balance[0]`** – перший елемент масиву, **`balance[99]`** – останній елемент масиву.

14.3 Оголошення та ініціалізація

Оголошений на внутрішньому рівні, масив не ініціалізований. Оголошений на зовнішньому рівні – ініціалізований нульовим значенням відповідно типу.

Список ініціалізації задають у фігурних дужках, тип літералів повинен співпадати з базовим типом або може бути неявно приведений до нього. Якщо кількість елементів списку ініціалізації співпадає з кількістю елементів – результат очевидний, наприклад:

```
int month[12]={31,28,31,30,31,30,31,31,30,31,30,31,30,31};
```

Обробку елементів масиву зручно виконувати з використанням циклів. Наприклад, для виведення на екран вмісту щойно ініціалізованого масиву:

```
for(int i=0;i<12;i++)printf("%d місяць: %d\n",i+1, month[i]);
```

результат складатиметься з 12 рядків такого вмісту:

```
1 місяць: 31
2 місяць: 28
…
12 місяць: 31
```

Якщо список ініціалізації виявиться коротшим, останні невизначені значення будуть ініціалізовані нулям, якщо довшим, виникне помилка, тому краще скористатись різновидом оголошення з порожніми квадратними дужками, наприклад:

```
double arr[]={1.,-8.6,9.14,3,15,0};
```

Для ініціалізованого таким способом масиву фрагмент роботи програми, що підраховує суму його елементів може виглядати так:

```
int dim=sizeof arr/sizeof(double), sum, i;
     for(i=0, sum=0; i<dim; i++)
                    sum+=arr[i];
     printf(" arr[%d], sum=%f\n", dim, sum);
```

Оскільки опрацювання усіх елементів масиву можна здійснювати тільки у циклі, то заповнення з клавіатури, виведення на екран, аналіз кожного елементу як правило реалізують з використанням циклів.

Заповнення масиву з клавіатури. Якщо заповнення масиву заплановане у процесі роботи програми, його потрібно спочатку оголосити. Розмір масиву задають за допомогою константного виразу або з використанням макроконстанти або оголошенням константи до оголошення масиву.

Після оголошення масиву з максимально очікуваним розміром, можна увести з клавіатури реальний розмір, з яким заплановано працювати.

Наприклад, заповнення елементів масиву з клавіатури:

```
#define MAX 100
int main()
{
  double arr[MAX];
  int i, n;
  printf("enter dimension\n");
  scanf("%d", &n);
  printf("enter %d array elements\n", n);
      for(i=0; i<n; i++)
              scanf("%lf", &arr[i]);
. . .
}
```

Тут для оголошення масиву використано макроконстанту **MAX** за допомогою директиви **#define.** Після оголошення масиву уведено значення **n** для визначення реальної розмірності масиву, який потрібно заповнити з клавіатури значеннями відповідно базового типу.

Інколи для перевірки коректності роботи програми використовують так звані псевдовипадкові числа. Масив можна заповнити такими числами, наприклад:

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#define MAX 100
int main()
{
  double arr[MAX];
  int i, n;
  srand(time(NULL)));
  printf("enter dimension\n");
  scanf("%d", &n);
```

```
        for(i=0; i<n; i++){
            arr[i]=rand();
            printf("%lf ", arr[i]);
    }
}
```

Як показано у наведеному прикладі значення чергового елементу одразу після його генерації потрібно виводити на екран для контроля коректності при подальшій роботі з масивом.

Словник. Масив, розмір, вимір, розмірність масиву, одновимірний масив, елемент, базовий тип, індекс, індексний вираз.

Завдання на СРС: Розв'язування задач по роботі з одновимірними масивами [7,с.110, с.136, 8, с.50]

Контрольні запитання

1. Що таке масив? Для чого дані структурують?
2. Надайте синтаксис оголошення масиву, поясніть семантику.
3. Що таке індекс? Індексний вираз? Які вимоги до коректності формування індексних виразів в С?
4. Як можна ініціалізувати одновимірний масив?
5. За допомогою якої конструкції мови можна заповнити масив з клавіатури? Вивести на екран? Заповнити випадковими значеннями?
6. Що можна сказати про ідентифікатор масиву?

Лекція №15

Тема лекції: Алгоритми роботи з одновимірними масивами

План лекції

1. Звертання до елементів масиву через індекси та через вказівники
2. Сортування елементів масиву

Зміст лекції

Типи задач можна поділити на декілька категорій. Наприклад, пошук елементів з наданою ознакою, пошук максимального, сортування, перетворення, зсув елементів, виділення частини, зміна розмірів, тощо.

15.1 Звертання до елементів масиву через індекси та через вказівники

Задача пошуку елементів з наданою ознакою може бути сформульована по-різному.

Для виконання певної дії над усіма елементами (підрахунок кількості, суми, тощо) з наданою ознакою, потрібен прохід по всьому масиву, аналіз кожного елементу. Такі задачі краще реалізувати з використанням **for**.

Наприклад, порахувати суму елементів, абсолютне значення яких менше деякого числа

```
double arr[MAX];
//уведення розмірності і вмісту масиву з клавіатури
for(i=0, sum=0; i<n; i++)
    if(fabs(arr[i])<eps)
          sum+=arr[i];
```

Задачі визначення наявності такого елементу не вимагають проходу усього масиву і підрахунок їх можна реалізувати і з використанням **while**.

Наприклад, наведений нижче фрагмент програми визначає чи містить масив цілих чисел хоча б одне парне:

```
int arr[MAX], i, n;
//уведення розмірності і вмісту масиву з клавіатури
i=0;
while(i<n&&arr[i]%2)i++;
if(i!=n)
     printf("yes, this array contains an even number\n");
```

Для пошуку потрібного елемента інколи достатньо запам’ятовувати тільки його індекс. Наприклад, знайти максимальний елемент масиву дійсних чисел і поміняти його з першим.

```
int imax, i;
//пошук індексу максимального елемента
for(i=0, imax=0; i<n; i++)
      if(arr[i]>arr[imax])imax=i;
//обмін
      double d=arr[0]; arr[0]=arr[i]; arr[i]=d;
```

Достатньо розповсюджені на практиці програми, що реалізують алгоритми зсуву, у тому числі і циклічного, елементів масиву.

Наприклад, необхідно реалізувати циклічний зсув елементів масиву на одну позицію вліво. У результаті кожному **i**-тому елементу потрібно буде присвоїти значення **i+1**-го, а нульовий елемент стане останнім. Фрагмент такої програми може виглядати так:

```
double arr[10]; int i, n=10;
// заповнення масиву дійсних чисел …
double d=arr[0];
for(i=0; i<n-1; i++)
      arr[i]=arr[i+1];
arr[n-1]= d;
```

Тут змінна **d** тимчасово зберігає значення нульового елементу.

15.2 Сортування елементів масиву

Для реалізації будь-якого з алгоритмів сортування «на місці», тобто без використання додаткових масивів, потрібні як мінімум два вкладених цикли: внутрішній забезпечує черговий прохід по масиву, а зовнішній визначає границі відсортованості. Наприклад, реалізацій алгоритму сортування методом обміну з прапорцем може виглядати так:

```
double arr[MAX];
int i,k,n,fl=1;
printf("enter dimention\n");
scanf ("%d ",&n);
printf("enter %d double numbers\n", n);
for(i=0;i<n;i++)
      scanf("%lf", &arr[i]);
k=n-1;
 while((fl!=0)&&(k>0)){
      fl=0;
      for(i=0;i<k;i++)
           if(a[i]>a[i+1]{
                temp=a[i]; a[i]=a[i+1];
                a[i+1]=temp; fl=1;
           }
      k--;
      }
```

Деякі задачі вимагають зміни розміру масиву. Якщо масив не динамічний, змінити можна лише розмір задіяної у задачі частини масиву. Тобто під масив виділяють деякий апріорі відомий максимальний розмір, вводять поточний розмір, як у наведених раніше прикладах, а потім, під час роботи програми у разі потреби цей поточний розмір змінюють.

Наприклад, нехай потрібно видалити із масиву усі елементи, абсолютне значення яких менше за деяке число **eps** зі зсувом тих, що залишились і зміною розміру.

```
double arr[MAX];
int i,k,n,j;
printf("enter dimention and epsilon\n");
scanf ("%d %lf ",&n, &eps);
printf("enter %d double numbers\n", n);
for(i=0;i<n;i++)
      scanf("%lf", &arr[i]); . . .
for(i=0, j=0;i<n;i++)
      if(fabs(arr[i])>eps){
           arr[j]=arr[i];
           j++;
      };
      n=j-1;
for(i=0;i<n;i++)
      printf ("%f", arr[i]);
printf("\n");
```

Після модифікації масиву, його вміст обов'язково потрібно вивести, адже це і є результатом роботи програми .

Реалізація деяких алгоритмів передбачає наявність декількох масивів, наприклад, реалізація алгоритму сортування злиттям неможлива без двох масивів одночасно, реалізація алгоритму зсуву одразу на декілька позицій, тощо.

У розглянутому далі прикладі потрібно сформувати третій масив із двох на основі тільки тих елементів із масиву **a**, яких немає в **b**.

```
double a[MAX], b[MAX], c[MAX];
int i, k, n, j, l;
// заповнення масивів дійсних чисел a i b …
for(i=0, l=0;i<n;i++){
      for(j=i, k=0;j<n;j++)
           if(a[i]==a[j])k++;
      if(!k)c[l++]=a[i];
}
```

Словник. Зсув, сортування, базові алгоритми сортування вставкою, методом вибору, методом обміну.

Завдання на СРС: Розв’язування задач по сортуванню одновимірних масивів[7,с.110, 8, с.60]

Контрольні запитання

1. Який зв'язок між значеннями індексу елемента в масиві і його адресою в пам'яті?
2. Яким циклом краще реалізувати прохід по всьому масиву і чому?
3. Які типи задач по роботі з масивами краще реалізувати з використанням циклу **`while`**?
4. Ідея базового алгоритму сортування методом обміну та обміну з прапорцем.

Лекція №16

Тема лекції: Двовимірні масиви

План лекції

1. Багатовимірні масиви
2. Двовимірні масиви
3. Визначення розміру пам'яті для двовимірних масивів
4. Приклади роботи з двовимірними масивами

Зміст лекції

16.1 Багатовимірні масиви

Масиви бувають одновимірні, двовимірні і багатовимірні. Найчастіше у програмуванні використовують одновимірні і двовимірні, дуже зрідка – тривимірні масиви. Вимірність масиву не обмежена можливостями мови, їх не використовують просто тому, що такий масив важко уявити і з ним незручно працювати програмісту. В мовах C, C++ **N**-вимірний масив розглядають як одновимірний масив із **N-1**- вимірних масивів.

Синтаксис оголошення N-вимірного масиву:

```
тип ідентифікатор[розм1][розм2] … [розмN]
```

Вимірність масиву визначена кількістю пар дужок в оголошенні масиву. *Розмірність* визначає максимальну кількість елементів по даному виміру і визначені у квадратних дужках, що відповідають даному виміру в оголошенні масиву. *Розмір* масиву визначає загальну кількість елементів масиву і дорівнюватиме добутку усіх розмірностей даного масиву. Основні принципи роботи з багатовимірними масивами можна дослідити на двовимірних масивах.

Двовимірні – *матриці* – можна уявити собі як прямокутну матрицю або таблицю, кожна клітинка в якій містить елемент певного типу. Його положення можна задати двома координатами – індексами – порядковими номерами рядка і стовпчика, на перетині яких цей елемент розташований.

Оскільки в C(C++) двовимірний масив представлений як одновимірний масив одновимірних масивів, в його оголошенні у квадратних дужках задають дві розмірності.

16.2 Двовимірні масиви

Синтаксис оголошення двовимірного масиву:

тип ідентифікатор[конст.вираз1][конст.вираз2]

Тут **тип** – це базовий тип елементів масиву, **конст.вираз1** визначає кількість рядків, **конст.вираз2** – кількість стовпчиків матриці, якій відповідає графічне уявлення про двовимірний масив.

Приклади оголошення:

```
#define N_max 853
#define M_max 157
...
int sample[N_max][M_max]; // з викор. макроконстант
int a[100][50];          // з використанням констант
const int t=5, k=8;
float wer[2*t+k][2*t+k]; // з викор. константних виразів
```

Елементи двовимірних масивів розташовані у пам'яті підряд як єдиний одновимірний масив по рядках – у порядку більш швидкої зміни другого індексу. І взагалі елементи масивів з вимірністю більше одиниці розташовані у пам'яті так, що *найскоріше змінюється самий правий індекс.*

З таким порядком розташування у пам'яті пов’язаний і синтаксис виразу ініціалізації масиву списком констант, як це показано для масиву дійсних чисел розмірністю 2x3:

```
float w[2][3]={{2.1, 3.4, 4.5},{5.0, 6.4, 3.9}};
```

У випадку відсутності потрібної кількості констант для ініціалізації усіх елементів масиву, вони будуть доповнені нулями .

Розглянутий вище вираз ініціалізації рівнозначний такому:

```
float w[][3]={ {2.1, 3.4, 4.5}, {5.0, 6.4, 3.9}  };
```

Опущений може бути тільки перший константний вираз. Тобто кількість рядків може залежати від списку ініціалізації. У загальному випадку, яка би не була кількість вимірів, можна не задавати тільки самий лівий вимір.

Звертання до елементів двовимірного масиву задається так само як до одновимірного, але з використанням двох індексів, кожний з яких розташований у окремих квадратних дужках.

Синтаксис звернення до елементу двовимірного масиву:

ідентифікатор[індекс1][індекс2];

Індекси – вирази або константні вирази, які можуть приймати цілі значення. Обидва індекси самого першого елемента дорівнюють нулю.

a[0][0] – індекс задано як константу,
s[i][j] – індекс задано як змінну,
w[4*p][3+t] – індекс задано як вираз.

Так само і для масивів більшої вимірності – кількість дужок залежить від кількості вимірів масиву.

16.3 Визначення розміру пам'яті для двовимірних масивів

Об'єм зайнятої пам'яті в байтах для двомірного масиву обчислюється за формулою:

Кількість байтів = sizeof(тип)*конст_вираз1*конст_вираз2

Оскільки двовимірний масив у пам'яті розташований як одновимірний, схему розташування елементів масиву можна представити так.

```
int w[2][5]={{2, 3, 5},{5, 4, 9}};
```

2	3	5	0	0	5	4	9	0	0

Отже, потрібно пам'ятати, що якщо оголошено масив з великими значеннями розмірностей по всім вимірам, наприклад, **N**=10, **M**=10, а реально він заповнений з деякими меншими розмінностями **n**=3, **m**=3, то у пам'яті вільні місця заповнені корисною інформацією не будуть.

16.4 Приклади роботи з двовимірними масивами

Оголошення і заповнення двовимірного масиву підпорядковане тим самим правилам, що і одновимірного. Тільки потрібні два вкладених цикли.

Наприклад, заповнення двовимірного масиву для виконання робіт комп'ютерних практикумів 4 і 5 потрібно виконати приблизно так:

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <ctype.h>
#define M 100
#define N 100

int main(){
  double arr[M][N];
  int i, n, m;   char ch;
  srand(time(NULL));
  printf("enter dimension n and m\n");
```

```
  scanf("%d%d", &n, &m);
  printf("use random values?(Y/N)\n");
  scanf("%c", &ch);
  if (toupper(ch)==’Y’){
//заповнення випадковими числами
         for(i=0; i<m; i++)
              for(j=0; j<n; j++)
                   arr[i][j]=rand()%100*0.1-5.;
    }
 else{
//заповнення з клавіатури
         printf("enter %d array elements\n", n*m);
              for(i=0; i<m; i++)
                   for(j=0; j<n; j++)
                        scanf("%lf", &arr[i][j]);
   }
//виведення на екран у вигляді прямокутника
    for(i=0; i<m; i++){
         for(j=0; j<n; j++){
              printf("%6.2f ", arr[i][j]);
              }
          printf("\n");
    }
return 0;
}
```

Потрібно звернути увагу, що після уведення значень елементів масиву, його варто вивести саме у формі правильного прямокутника, щоб легко було у подальшому аналізувати отримані результати.

Наступна задача – пошук максимального елемента матриці. Для цього також потрібні два вкладених цикли. Після заповнення масиву цей фрагмент виглядатиме так:

```
int im,  jm; . . .
    for(i=0, im=0, jm=0; i<m; i++)
         for(j=0; j<n; j++)
              if(arr[i][j]>arr[im][jm]){im=i; jm=j;}
 printf("max:arr[%d][%d]=%6.2f\n ", im, jm, arr[im][jm]);
```

Окремим випадком прямокутного масиву може бути квадратний масив, коли кількість рядків і стовпчиків однакова.

У цьому випадку можна говорити про діагональні елементи масиву. Матриця, що відповідає такому масиву, має дві діагоналі – головну (з лівого верхнього кута у правий нижній) і побічну (з правого верхнього).

Якщо квадратний масив має розмірність **N*N**, кількість елементів на головній і побічній діагоналях дорівнюватиме також **N**. Індекси **i** та **j**

елементів, розташованих на головній діагоналі задовольняють умові **i==j,** а на побічній : **i==N-j-1.**

При роботі з діагональними елементами їх можна розглядати як одновимірний масив. Наприклад, фрагмент програми, що підраховує середнє арифметичне елементів головної діагоналі, виглядатиме так:

```
double arr[N][N], sum; . . .
   for(i=0, sum=0; i<n; i++) sum+= arr[i][i];
   printf("average=%6.2f \n", sum/n)
```

Словник. Багатовимірний масив, двовимірний масив, діагональний елемент.

Завдання на СРС: Розв'язування задач по роботі з двовимірними масивами [8,с.51, 4,с.139]

Контрольні запитання

1. Як оголосити двовимірний масив? У яких випадках можна опустити визначення кількості елементів в оголошенні масиву?
2. Як заповнити усі елементи масиву з клавіатури?
3. Як вивести на екран у вигляді таблиці?
4. Що буде виведено на екран в результаті виконання такого фрагмента коду:

```
float mas[3][5]={{2.0},{4.5,8.3},{7.0,1.0,5.5,7.8}};
int t=sizeof(mas);
printf( "mas[3][5]={{2.0},{4.5,8.3},{7.0,1.0,5.5,7.8}}
         займає %d  байтів \n", t);
```

Лекція №17

Тема лекції: . Оголошення, ініціалізація та уведення-виведення символьних рядків

План лекції

1. Загальна характеристика символьної інформації
2. Варіанти оголошення та ініціалізації рядків
3. Варіанти організація уведення-виведення символів за допомогою стандартних бібліотечних функцій
4. Функції класифікації і перетворення символів

Зміст лекції

17.1 Загальна характеристика символьної інформації

Тип **char** (скорочення від **charakter** – літера) віднесено до цілих базових типів. У більшості реалізацій С займає один байт і містить **ASCII**-код символу.

Символьна константа – це один або декілька охоплених апострофами символів. Константа може складатися з одного символу або послідовності символів, які починаються з символу **** (обернений слеш) і називаються керуючими або **escape**-послідовностями. Код символу у вісімковій, десятковій або шіснадцятковій системах числення також може бути заданий у вигляді **escape**-послідовності.

Приклади оголошення і ініціалізації символьних змінних

```
char a='a', b='5', sym='*', s;
char a='\n';      // escape-послідовність
char a='\235';    // десятковий код символу
char b='\023';    // вісімковий код символу
char c='\x23';    // шіснадцятковий код символу
```

17.2 Варіанти оголошення та ініціалізації рядків

Для оголошення символьних рядків спеціального типу немає. Символьний рядок представлений як спеціальний різновид одновимірного масиву, який складений із елементів типу **char**. Останній елемент рядка завжди містить так званий нуль-символ **'\0'**, який має код **0**.

Рядковий літерал – константа типу символьного рядка – послідовність довільних символів в подвійних лапках, обов'язково містить останній **'\0'**.

Приклади рядкових літералів: **"some information", "words"**

Приклади оголошення і ініціалізації символьних рядків:

```
char str1[20];
str2[] = "1234", str3[]={'w','o','r','d','\0'};
char *strp="next sentence";
```

Оскільки рядок – це масив, для роботи з його вмістом можна застосовувати як індексну форму, так і форму доступу з вказівниками, наприклад:

```
char str1[20], *p=&str1[0],  *q=&str1[19];
```

У даному прикладі оголошений масив символів і два вказівники – на перший і останній його елементи.

17.3 Варіанти організація уведення-виведення символів за допомогою стандартних бібліотечних функцій

Для уведення-виведення символів використовують функції стандартної бібліотеки, прототипи яких містить заголовковий файл **<stdio.h>**. Це функції **putchar, getchar, printf** та **scanf.**

Прототип функції введення символу:

```
int getchar ( void );
```

Повертає код уведеного символу, доповнений до **int** нулями у старшому байті, якщо все нормально. Якщо трапилась помилка – повертає **EOF**.

Прототип функції виведення символу:

```
int putchar( int character );
```

Параметр функції – це символ, який потрібно вивести, повертає його код, якщо все успішно, або **EOF.**

Уведення символів полягає у занесенні кожного у буфер клавіатури аж до натискання клавіши **ENTER**, після чого уведену інформацію уже не можна

корегувати і програма виконується далі. Кожний уведений символ буде відображений на екрані.

Приклад використання:

```
char character;
printf("введіть символ, вихід - символ точки:");
do{
  character = getchar();              // ввести символ
   putchar (character);          // вивести цей символ
} while (character != '.');    // поки це не точка
```

Програма вводить з клавіатури рядок по одному символу до крапки, а потім виводить його на екран.

Для форматного введення і виведення символів використовують функції **printf** та **scanf** зі специфікатором **%c**. Наприклад:

```
char character;
scanf_s(" %c", &character);
printf("character = %c\n", character);
```

Ці функції були розглянуті раніше.

17.4 Функції класифікації і перетворення символів

Заголовковий файл **<ctype.h>** містить групу функцій, призначених для класифікації символів та виконання простих операцій над ними.

Функції, назва яких починається з **is** призначені для визначення приналежності даного символу до якоїсь кваліфікаційної групи і повертають значення 0 або 1 (**false**/**true**).

Функції, назва яких починається з **to** призначені для реалізації тривіального перетворення даного символу і повертають результат цього перетворення.

Прототипи деяких з цих функцій та їх семантика зведені до наступної таблиці 17.1.

Таблиця 17.1 Деякі функції класифікації і перетворення **<ctype.h>**

Прототип	Призначення
`int isalnum(int c);`	символ c є ASCII-кодом латинської букви або цифри ('a'-'z', 'A'-'Z', '0'-'9')
`int isalpha(int c);`	символ c є ASCII-кодом латинської букви('a'-'z', 'A'-'Z')
`int isascii(int c);`	символ c є ASCII-кодом (0-127)
`int isdigit(int c);`	символ c є ASCII-кодом десяткової цифри ('0'-'9')

int iscntrl(int c);	символ c є ASCII-кодом керуючого знаку (0x7f або 0x00-0x1f)
int islower(int c);	символ c є ASCII-кодом малої букви ('a'-'z')
int isprint(int c); **int isgraph(int c);**	символ c є ASCII-кодом видимого знаку (0x20-0x7e), (крім пропуска 0x20 для isprint)
int ispunct(int c);	символ c є ASCII-кодом розділового знаку
int isspace(int c);	символ c є ASCII-кодом пропуска
int isupper(int c);	символ c є ASCII-кодом великої букви ('A'-'Z')
int isxdigit(int c);	символ c є ASCII-кодом шіснадцяткової цифри ('0'-'9', 'a'-'f', 'A'-'F')
int toascii(int c);	перетворює цілі числа в ASCII-коди
int tolower(int c);	перетворює великі латинські букви у маленькі
int toupper(int c);	перетворює маленькі латинські букви у великі

Приклад використання:

```
char character;
character=getchar();
while( !isdigit(character)){
   if(islower(character))
      putchar(toupper(character));
    else
      putchar(character);
   character=getchar();
}
```

Наведений фрагмент програми призначений для зчитування з клавіатури рядка символів до першої цифри, а потім виведення його з заміною букв нижнього регістра на верхній.

Словник. Символьний рядок, рядковий літерал, нуль-символ, буфер уведення-виведення, стандартні потоки, макроконстанта **EOF**.

Контрольні запитання

1. Як оголосити символьний рядок? Як оголосити рядок, якщо інформацію до нього потрібно увести з клавіатури?
2. Що означає і для чого призначений символ **'\0'**?
3. Яке значення повертає стандартна бібліотечна функція **getchar()**?
4. Для чого призначені функції, оголошені у **<ctype.h>?**

Лекція №18

Тема лекції: . Робота з рядками як з масивами символів

План лекції

1. Варіанти організація уведення-виведення символьних рядків за допомогою стандартних бібліотечних функцій
2. Робота з рядками з використанням індексів.
3. Робота з рядками з використанням вказівників

Зміст лекції

18.1 Варіанти організація уведення-виведення символьних рядків за допомогою стандартних бібліотечних функцій

Для введення і виведення рядків використовують функції **gets, puts, printf** та **scanf.**

Для буферизованого потокового уведення і виведення рядків використовують функції стандартної бібліотеки, прототипи яких оголошені у заголовковому файлі **<stdio.h>**. Це функції **gets()** та **puts().**

Прототип функції **gets** наступний**:**

```
char * gets( char * string );
```

Функція зчитує символи із стандартного потоку – з клавіатури за умовчанням – до буферу **string** поки не отримає символ нового рядка `'\n'` або кінця файлу **EOF**, який вона не зберігає. В кінець отриманого набора символів функція ставить символ `'\0',` чим завершує формування рядка.

Розмір буферу (символьного масиву) повинен бути достатнім для розташування рядка. За цим має стежити програміст.

Повертає функція вказівник на той самий буфер, якщо все успішно, або **NULL**, якщо трапилась помилка або був досягнутий кінець файлу (тоді вміст масиву – не визначений).

Функція **gets** зчитує з потоку усі символи, у тому числі і пропускові.

Наприклад:

```
#include <stdio.h>
…
char str[80];
    printf (" введіть рядок ");
    gets(str);
```

Функція **puts** має прототип:

```
int puts( const char * string );
```

Виводить до стандартного потоку (на екран за умовчанням) вміст рядка, який замість символу '\0' завершує символом '\n', тобто обов'язково переводить курсор до наступного рядка на екрані.

Приклад використання:

```
char str[80]= "this is first sentence ";
    puts(str);
    puts("this is next sentence ");
```

У результаті роботи наведеного фрагменту на екран будуть виведені два речення, друге – з нового рядка.

Якщо функція відпрацювала нормально – вона повертає невід'ємне число, якщо трапилась помилка – **EOF.**

Функцію форматного введення рядка **scanf** використовують зі специфікатором **%s.** Вона працює так само, як і для введення змінної будь-якого іншого із припустимих типів, але потрібно врахувати, що ідентифікатор рядка сам по собі є адресою, тому амперсант **&** перед ним ставити не потрібно. Крім того, пропуск або інший пробільний символ у середині рядка функція сприйме як його закінчення і решту рядка не вводитиме. Тобто замість речення функція зчитає тільки його перше слово.

Функція форматного виведення рядка **printf** також використовує специфікатор **%s.** Вивести можна як слово, так і речення (на відміну від **scanf**). Після виведення рядка, курсор залишиться після останнього виведеного символу (на відміну від **puts**).

Для наступного фрагменту:

```
char sent[100];
     puts("enter sentence:");
     scanf("%s", sent);
     printf("%s - THE END...\n", sent);
```

Результат роботи на екрані буде таким:

```
enter sentence:
first second third
first - THE END...
```

У наведеному прикладі видно, що з клавіатури до буферу зчитане було тільки перше слово

18.2 Робота з рядками з використанням індексів

Оскільки рядок – це масив, для роботи з його вмістом можна застосовувати індексну форму, наприклад:

```
#include <stdio.h>
…
char str1[80],str2[80];
    printf (" введіть рядок ");
    gets(str2);
    int i;
    for(i=0;str2[i]!='\0';i++)  str1[i]=str2[i];
    str1[i]='\0';
```

У наведеному прикладі вміст уведеного з клавіатури рядка **str2** копіюють до рядка **str1.** Оскільки з клавіатури рядок був введений за допомогою функції **gets**, після останнього символа в ньому знаходиться нуль-символ і цим можна скористатись для визначення умови продовження циклу.

Інший приклад – визначення кількості слів у реченні, уведеному з клавіатури.

```
char str[80];
    printf (" введіть рядок ");
    gets(str);
    int i,count;
    for(i=0,count=0;str[i]!='\0';i++)
        if(str[i]==' ')&&(str[i+1]!=' ')count++;
     printf (" рядок містить %d слів\n", count);
```

У наведеному прикладі використаний той факт, що слова обов'язково розділені пропусками, навіть якщо у реченні використані розділові знаки. Врахований також варіант, якщо між словами більше, ніж один пропуск.

18.3 Робота з рядками з використанням вказівників

У той же час назва рядка – константний вказівник на його перший символ, отже для роботи з його вмістом можна застосувати і форму доступу з вказівниками. Той же приклад з вказівниками виглядатиме так:

```
char str1[80],str2[80], *p, *q;
  printf (" введіть рядок ");
  gets(str2);
  for(p=str2, q=str1;*p!=0;p++, q++) *q=*p;
  *q='\0';
```

У наведеному прикладі, як і в попередньому, нуль-символ до нового рядка був вставлений окремо, оскільки із старого рядка він не був скопійований у циклі.

Для роботи з рядками найчастіше використовують саме форму доступу з вказівниками.

Завдання на СРС: Розв’язування задач по роботі з символьними рядками [8, с.66, 7, с.44, 6,с.159]

Контрольні запитання

1. У яких випадках рядок краще уводити з використанням функції **`gets()`**?
2. У яких випадках рядок краще уводити з використанням функції **`scanf()`**?
3. У чому різниця між виведенням рядка за допомогою функцій **`puts()`** та **`printf()`** ?

Лекція №19

Тема лекції: Бібліотечні функції для роботи з символьними рядками

План лекції

1. Функції операцій над рядками
2. Функції перетворень рядків символів у числа та зворотних перетворень

Зміст лекції

19.1 Функції операцій над рядками

Заголовковий файл **<string.h>** містить групу функцій, призначених для операцій над символьними рядками. Використання цих функцій може передбачати перетворення рядка або використання декількох рядків, тому важливо враховувати те, що сумарний рядок має бути достатнього розміру, що рядки не повинні перетинатися (кваліфікатор **restrict** в С99) у пам'яті і що кожний рядок повинен закінчуватись нулем, інакше функція не працюватиме правильно.

Ідентифікатори всіх цих функцій починаються з префіксу **str.** Протопипи і призначення функцій, які часто використовують і які потрібні для виконання робіт компютерного практикуму 6, зведені до таблиці 19.1.

Таблиця 19.1 Деякі функції роботи з символьними рядками **<string.h>**

Прототип	Призначення
`char * strchr(const char *string, int c);`	знаходить перше входження символу `c` до рядка `string` і повертає вказівник на нього або `NULL`, якщо такий не знайдено
`char * strrchr(const char *string, int c);`	знаходить останнє входження символу `c` до рядка `string` і повертає вказівник на нього або `NULL`, якщо такий не знайдено
`char *strcat(char *dst, const char *src);`	долучає копію рядка `src` разом з `/0` до рядка `dst` так, що перший символ `src` заміщує останній `/0` в `dst`
`char *strncat(char *dst, const char *src, size_t length);`	аналог `strcat` для перших `length` символів `src` (або до `/0`, якщо рядок коротший за `length`)
`int strcmp(const char *a, const char *b);`	виконує лексикографічне порівняння двох рядків (-1,0,1)
`int strncmp(const char *a, const char * b, size_t length);`	аналог `strcmp` для перших `length` символів
`char *strcpy(char *dst, const char *src);`	виконує копіювання рядка `src` до масиву `dst`
`char *strncpy(char *dst, const char *src, size_t`	аналог `strcpy` для перших `length` символів

length);	
size_t strlen(const char *str);	рахує довжину рядка (без /0)
char *strstr(const char *s1, const char *s2);	знаходить перше входження символів рядка `s2` (без /0) до рядка `s1` і повертає вказівник на знайдений підрядок або `NULL`, якщо такий не знайдено
size_t strspn(const char *s1, const char *s2);	рахує кількість букв із `s2`, які зустрічаються на початку `s1`, навіть якщо порядок їх слідування не співпадає
char *strtok(char *source, const char *delimiters)	у результаті серії викликів розбиває рядок `source` на лексеми, розділені знаками із `delimiters`

Функція **strlen** дозволяє визначити довжину рядка. Наприклад, потрібно скоротити рядок до 40 символів, якщо він довший. Це можна зробити так:

```
#include <string.h>
. . .
char str[80];
      gets(str);
      if(strlen(str)>40) *(str+40)= '\0';
```

Приклад копіювання вмісту одного рядка **(str1)** до іншого **(str2):**

```
#include <string.h>
. . .
char str1[80], str2[80];
      gets(str1);
      if(strcpy(str2,str1);
```

Функція **strcmp** виконує лексикографічне порівняння, повертає **0** якщо рядки однакові за довжиною і вмістом, менше нуля **(-1),** якщо перший менше другого і більше нуля **(1),** якщо навпаки. Порівняння відбувається за кодом першого не співпадаючого символу, або довжиною. Наприклад якщо **s1="enter"** і **s2="en", strcmp(s1, s2)** поверне 1, а якщо **s1="enterrr"** і **s2="entesr", strcmp(s1, s2)** поверне -1, тому що код **r** менше за код **s.**

Результат коректний тільки для латинських букв, адже саме вони впорядковані за алфавітом. Функція чутлива до регістру.

Функцію **strtok,** яка виконує розбиття символьного рядка на окремі лексеми, розділені у тексті рядка конкретними символами, використовують у циклі. Початковий рядок зміниться, оскільки після кожної виділеної лексеми в текст будуть вставлені нуль-символи.

19.2 Функції перетворень рядків символів у числа і зворотних перетворень

Заголовковий файл **<stdlib.h>** містить групу функцій, призначених для перетворення символьних рядків у числа. Прототипи і семантика цих функцій описані у таблиці 19.2.

Таблиця 19.2 Деякі функції перетворення у числа **<stdlib.h>**

Прототип	Призначення
`double atof(const char *s);`	Перетворює початкову частину рядка у число типу double; очикуваний формат `[+|-]digits[.][digits][(E|e)[+|-]digits]` повертає відповідне число або 0.0
`float atoff(const char *s);`	Перетворює початкову частину рядка у число типу `float`
`int atoi(const char *s);`	Виділяє у рядку `s` перше ціле десяткове і перетворює його у змінну типу `int`, кінець – перший нецифровий символ
`long atol(const char *s);`	Виділяє у рядку `s` перше ціле десяткове і перетворює його у змінну типу `long`, кінець – перший нецифровий символ
`double strtod(const char *str, char **tail);`	Розширений варіант `atof`, другий параметр – адреса першого символа за числом
`float strtodf(const char *str, char **tail);`	Розширений варіант `atoff`, другий параметр – адреса першого символа за числом
`long strtol(const char *s, char **ptr,int base);`	Розширений варіант `atol`, другий параметр – адреса першого символа за числом, третій – система числення від 2 до 36, якщо 0, то розшифровка відповідно загальних правил 10/8/16

Функції **atof, atoff, atoi, atol** працюють тільки з початком рядка, тому, якщо числа знаходяться в середині тексту, їх потрібно спочатку відшукати і вжити заходів для можливості застосування цих функцій.

Функції типу **strtol** мають другий параметр – адресу першого символа за виділеним числом, яку визначає сама функція і яку можна використовувати для подальшого аналізу.

Приклад використання розлянутих функцій. Нехай з клавіатури уводиться речення, серед слів якого можуть зустрічатися дійсні додатні числа. Написати програму, яка знаходить значення найбільшого з цих чисел або виводить -1. Відомо, що кількість символів речення не перевищує 100, слова речення розділені пропусками і, можливо, комами.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100
int main(){
     char sent[MAX],*pw, delim[]=" ,";
     // роздiлювачi - пропуск i кома
     double temp, max=-1;
      //оскiльки усi можливi числа - додатнi
     puts("enter sentence:\n");
     gets(sent);
     pw=strtok(sent, delim); //видiлити першу лексему
do{
     if (isdigit(*pw)){ //якщо перший символ слова - цифра
          temp= atof(pw); //виконати перетвор. на дiйсне число
          if(temp>max) max=temp;
          }
     pw=strtok(NULL,delim); //видiлити наступну лексему
     }while(pw!=NULL);
printf("%lf\n", max);
return 0;
}
```

Функції зворотного перетворення – із числа в рядок символів – не входять до стандарту ANSI C, але представлені в більшості систем програмування на C та C++. Для їх використання необхідно також підключити **<stdlib.h>**. Це функції:

```
char *itoa(int num, const char *str, int base);
char* ltoa(long num, const char* str, int base);
char* ultoa(unsigned long num, const char* str, int base);
```

Ці функції відрізняються між собою тільки типом параметра **num** – цілого числа, яке потрібно перетворити. Всі три функції формують із числа **num** рядок символів, що відповідає запису цього числа в системі числення з основою **base**, і повертають вказівник на перший символ створеного рядка. Сформований рядок записується за адресою **str**. Функції **itoa()** та **ltoa()** записують від'ємні числа зі знаком мінус

Завдання на СРС: Розв’язування задач по роботі з символьними рядками[8,с.66, 7, с.44, 6,с.159]

Контрольні запитання

1. Який заголовковий файл потрібно підключити для обробки символьних рядків (копіювання, порівняння, тощо)?

2. Що означає термін «*лексикографічне порівняння*»?
3. Як склеїти два рядка? Що потрібно врахувати при цьому?
4. Як скоротити рядок?
5. Який заголовковий файл потрібно підключити для перетворення символьних рядків у числа?
6. Як вивести на екран окремі слова уведеного з клавіатури рядка у стовпчик?

Лекція №20

Тема лекції: Масиви символьних рядків та масиви вказівників

План лекції

1. Масиви рядків
2. Масиви вказівників на рядки
3. Збереження рядків у динамічній пам’яті

Зміст лекції

20.1 Масиви рядків

Рядки можна групувати у масиви, як і будь-які інші об’єкти мови. Але оскільки рядок сам по собі є одновимірний масив, хоча і специфічний, існують два різних підходи для роботи з таким масивом – як власне з масивом рядків, тобто фактично двовимірним масивом символів, і як з одновимірним масивом вказівників на окремі рядки. Принципова різниця полягає у тому, що в першому випадку рядки розташовані підряд, у другому – не обов’язково.

Синтаксис оголошення масиву рядків:

```
char ідентифікатор[вираз1][вираз2];
```

Тут **ідентифікатор** – це ідентифікатор масиву, **вираз1** визначає кількість рядків, **вираз2** визначає кількість символів у рядку.

Якщо масив буде заповнений пізніше, наприклад, з клавіатури, він може бути оголошений наприклад так:

```
char animals[5][10];
```

Кожний рядок такого масиву матиме фіксований розмір, але якщо він оголошений в середині функції і не ініціалізований, то не буде сприйматися компілятором як рядок, адже не матиме нуль-символа серед своїх елементів.

Якщо масив оголошений і одразу ініціалізований, перший вираз може бути відсутнім, наприклад:

```
char animals[][10]={"dog",cat","elefant","mouse","monkey"};
```

Таким масивом можна скористатись для розташування у ньому окремих лексем, наприклад слів, із речення, або сформувати речення із окремих рядків масиву, що задовільняють певній умові, тощо.

Наведений нижче фрагмент програми формує і виводить на екран речення із тих рядків масиву слів, що починаються з великої букви.

```
#include <stdlib.h>
#include <string.h>
#define MAX 10
int main()
{
char arr[MAX][MAX], sent[MAX*MAX]={"/0"};
. . .
     for(i=0; i<MAX; i++)
          if (isupper(arr[i][0]))// перша буква – велика
               strcat(sent, arr[i]);
     puts(sent);
     return 0;
}
```

Тут ініціалізація масива **sent** потрібна, щоб компілятор міг сприйняти його як рядок.

20.2 Масиви вказівників на рядки

Синтаксис оголошення масиву вказівників на рядки наступний:

```
char * ідентифікатор[вираз];
```

Тут **ідентифікатор** – це ідентифікатор масиву, **вираз** визначає кількість рядків, а кількість символів у рядку не визначена.

Такий масив повинен бути або одразу ініціалізований у статичній пам'яті, або під кожний з елементів цього масиву потрібно виділити динамічну пам'ять. Виділити пам'ять можна і після оголошення без ініціалізації.

Приклад оголошення з ініціалізацією:

```
char* animals[]={"dog",cat","elefant","krokodile","monkey"};
```

20.3 Збереження рядків у динамічній пам’яті

Для роботи з динамічною пам’яттю у стандарті мови С визначені чотири функції з прототипами у заголовковому файлі **<stdlib.h>.** Це **malloc, calloc, realloc, free.**

Прототип функції **malloc**:

```
void * malloc (size_t size);
```

Тут **size_t** – один з беззнакових цілих типів, **size** – розмір виділеної пам’яті у байтах. Повертає або адресу першого байту виділеної області, або **NULL**, якщо пам’ять не вдалося виділити.

Вміст комірок виділеної пам’яті функція не очищує.

Оскільки **malloc** повертає **void*,** для сумісності С і С++ програм, краще використовувати форму виклику цієї функції з явним приведенням типу:

```
pw=(char*)malloc(size);
```

Розмір потрібно задавати у байтах, тобто

```
size = <кількість елементів>* sizeof (<тип елементу>)
```

Приклад заповнення динамічного масиву рядків з клавіатури:

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

int main()
{
 char *mas[10];
 int i=-1;
     do{
        puts("enter next string < 20 characters");
        mas[++i]=(char*)malloc(20);
        gets_s(mas[i],20);
     }while(mas[i][0]!='\0');
 return 0;
}
```

Функція **calloc** також призначена для виділення пам’яті. Прототип:

```
void * calloc (size_t number, size_t size);
```

Тут **number** означає кількість елементів масиву, **size** – розмір кожного в байтах. На відміну від **malloc,** функція **calloc** очищує вміст комірок виділеної області.

Приклад використання:

```
double mas[num], *p=(double*)calloc(num, sizeof(double));
      if(p) //продовження роботи
```

Якщо виникає потреба у зміні розміру виділеної пам’яті, потрібно скористатися функцією **realloc**. Синтаксис:

```
void * realloc (void * ptr, size_t size);
```

Тут **ptr** вказує на адресу ділянки пам’яті, розмір якої потрібно змінити, а **size** – новий розмір в байтах. Повертає функція вказівник на ділянку зі зміненим розміром, або **NULL**, якщо цього не вдалося зробити. Інформація із

старої ділянки у разі успішного виділення пам’яті буде перенесена до нової. Стара ділянка може бути звільнена.

Приклад використання. Збільшити обсяг виділеної пам’яті вдвічі.

```
double* arr=(double*)realloc(mass,num*2);
if(arr){  mass=arr; . . .   }
else printf("memory error\n");
```

У розглянутому прикладі адреса нової пам’яті спочатку відмінна від адреси старої, адже немає гарантії успішності процесу виділення такого обсягу пам’яті.

Для звільнення пам’яті, виділеної будь-якою з розглянутих функцій, використовують функцію

```
void free (void*ptr);
```

Приклад використання.

```
free (mass);
```

У лекції були розглянуті варіанти оголошення та використання масивів символьних рядків, розташованих як у статичній, так і у динамічній пам'яті, а також розглянуті визначені в **stdlib.h** функції виділення та звільнення динамічної пам'яті.

Контрольні запитання

1. У чому відмінність оголошення масивів символьних рядків і двовимірних масивів символів?
2. Як можна ініціалізувати масив рядків?
3. Які переваги зберігання масивів символьних рядків у динамічній пам'яті?

Розділ 6. Функції та механізм передачі параметрів

Лекція №21

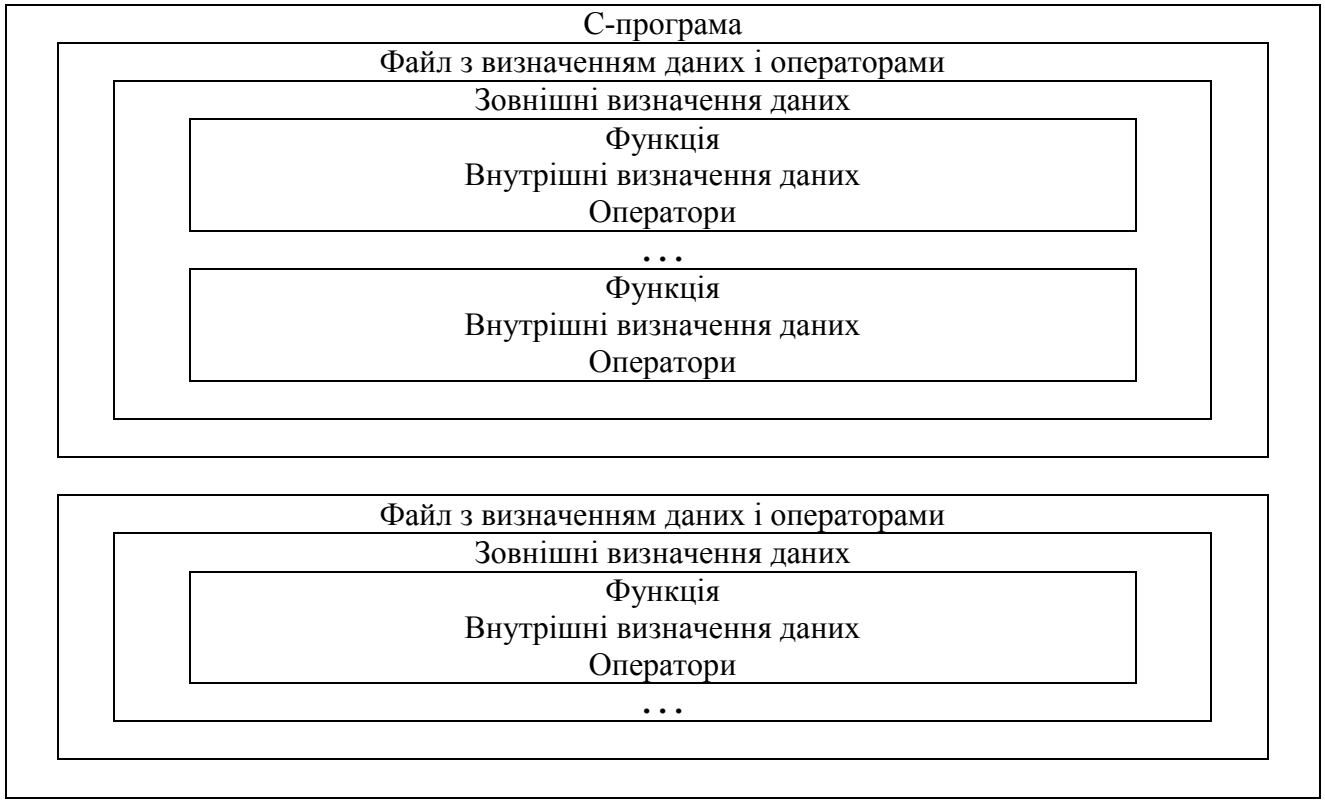
Тема лекції: Функції. Механізм передачі параметрів. Глобальні та локальні змінні

План лекції

1. Структура функції
2. Оголошення і виклик функції
3. Механізм передачі параметрів
4. inline–функції
5. Глобальні та локальні змінні

Зміст лекції

Вигляд програми мовою С схематично можна представити так:



21.1 Структура функції

Функції – основні складові програми мовою С.

Функція – логічно завершений іменований фрагмент програми, призначений для виконання деякої задачі. Унікальний ідентифікатор функції використовують для її виклику із іншої функції. В С функції формують окремий тип даних.

Використання функцій дозволяє один і той же фрагмент програми використовувати повторно, забезпечує кращу структурованість програми, що полегшує читання коду та його модифікацію.

У загальному випадку функція отримує дані при виклику через свої аргументи, виконує дії і повертає обчислене значення в місце виклику функції, де б цей виклик не знаходився. В якості окремого випадку функція може не мати аргументів і не повертати значення. При цьому вона виконує деякі дії, не пов'язані із зміною даних.

Функції можуть бути оголошені і описані в С тільки на глобальному рівні, тобто функція не може бути оголошена в середині іншої функції.

Окремі функції можуть бути відкомпільовані окремо зі створенням окремих об’єктних файлів (з розширенням **.obj**) і об’єднані у програму на етапі компонування.

Усяка функція повинна буті оголошена, визначена і викликана.

21.2 Оголошення і виклик функції

Оголошення (*заголовок*, *прототип*) повинно знаходитись до її виклику. Оголошень може бути декілька. Синтаксис оголошення:

```
тип ідентифікатор ([список формальних параметрів]);
```

тип – це тип, який функція повертає або тип функції. Це може бути будь-який із базових типів або їх модифікації, користувацький тип, вказівники на ці типи. Але функція не може повертати масив або іншу функцію, проте може повертати вказівник на масив або вказівник на функцію. Функція може повертати тип **void**, тобто нічого не повертати.

Ідентифікатор функції повинен відповідати усім вимогам до синтаксису ідентифікаторів та бути унікальним в межах даної програми.

Список формальних параметрів містить перелік оголошенн змінних через кому, які будуть використані в середині функції, або тільки список їх типів, якщо це окреме оголошення функції.

Визначення (опис, реалізація) функції включає її оголошення і тіло функції – оформлену як блок послідовність виконуваних і невиконуваних операторів функції.

Визначення функції завжди одне. В тексті програми може знаходитись як до, так і після опису функції, що її викликає. Синтаксис:

```
тип ідентифікатор ([список форм. параметрів])
                                { /* тіло функції*/ };
```

Приклад опису функції, що визначає регистр символу, переданого їй як параметр:

```
int func(unsigned char c){
      if (c>='A' && c<='Z')return 1;
           else  return 0;

 }
```

Функція повертає значення, якщо її виконання закінчується оператором **return,** що містить деякий вираз. Оператор **return** перериває виконання функції. Функція може містити декілька операторів **return**, а може не містити жодного і тоді повертати тип **void**.

Порядок і типи формальних параметрів повинні бути однаковими у визначенні функції і у всіх її оголошеннях. Тип формального параметра може бути будь-яким основним типом, структурою, об'єднанням, перерахуванням, вказівником або масивом.

Виконання функції починається з першого виконуваного оператора в момент виклику із іншої функції.

Синтаксис *виклику*:

```
ідентифікатор ([список фактичних параметрів]);
```

Типи фактичних аргументів при виклику функції повинні бути сумісні з типами відповідних формальних параметрів.

Наприклад:

```
int add(int x, int y){return x+y;}; //опис функції
//опис іншої функції
. . .
int i,var;
var=add(i,24); //виклик із іншої функції
```

21.3 Механізм передачі параметрів

Формальними називають параметри, які використовують у списку параметрів в оголошенні функції та її описі. Вони визначають типи змінних та порядок їх використання в алгоритмі, який реалізує функція.

Фактичними називають ті змінні або константи, які використовують в момент виклику функції. За типами і кількістю вони повинні бути узгодженими з формальними параметрами.

Механізм передачі параметрів наступний:

1) Послідовне обчислення значень виразів, заданих у виклику функції, тобто значень її фактичних параметрів.

2) Перетворення типу кожного фактичного параметра до відповідного формального параметру за правилами узгодження типів. Типи **`short`**, **`char`** обов'язково будуть перетворені до **`int`**, **`float`** до **`double`**.

3) Послідовне занесення до стеку отриманих значень. Стек – ділянка оперативної пам'яті, яка виділена для передачі параметрів даної функції.

4) Формальні параметри отримують значення із стека.

5) Оператори функції працюють з тими значеннями, які отримали формальні параметри.

6) Якщо функція повертає якесь значення, воно також буде занесене до стеку з відповідним перетворенням в разі потреби.

7) По закінченні функції, стек буде звільнено.

21.4 `inline`–функції

Такий різновид функцій вперше визначений в стандарті С99 і в С++. Це невеличкі функції, код яких компілятор може вставити безпосередньо в місце виклику.

Синтаксис оголошення таких функцій включає префікс **`inline`**, а самі вони як правило містять всього декілька виразів. Таке збільшення коду несуттєве порівняно з часом, що буде витрачений на передачу параметрів і передачу управління під час кожного виклику такої функції.

Приклад оголошення і виклику **`inline`**-функції, що обчислює максимум із двох чисел:

```
inline double max(double a,double b){
     return (a>b)?a:b;
};
int main(){
    double arr[N], m; int i; …
    for(i=1, m=arr[0]; i<N; i++)
```

```
                    m=max(m, arr[i];
    printf("max=%f\n",m);
return 0;
}
```

21.5 Глобальні та локальні змінні

Змінні, оголошені всередині функції, називають локальними, їх область видимості і область дії обмежена даною функцією. Змінні, оголошені як формальні параметри функції також локальні, оскільки видимі тільки в межах функції.

Змінні, оголошені на зовнішньому рівні, за межами усіх функцій, називають глобальними і вони будуть видимі і всередині функцій.

Область дії, облас видимості і час життя кожної змінної визначена класом пам'яті, до якого вона віднесена явно або неявно.

Словник. Функція, оголошення, прототип, сигнатура, визначення, опис, ідентифікатор, список параметрів, тіло функції, формальний параметр, фактичний параметр, передача параметрів, инлайн функції, глобальні і локальні оголошення.

Контрольні запитання

1. Опишіть структуру заголовка функції та правила оголошення формальних параметрів.
2. Що таке прототип функції? Де у програмі може бути розміщений прототип функції?
3. Яке значення може повертати функція?
4. Як мають бути узгоджені між собою формальні і фактичні параметри?
5. Скільки операторів **`return`** може бути в описі функції?
6. Чи завжди виклик функції приводить до передачі управління за адресою цієї функції?

Лекція №22

Тема лекції: Масиви та символьні рядки як параметри функції

План лекції

1. Одновимірні масиви як параметри функції.
2. Рядки як параметри функції.
3. Двовимірні масиви як параметри функції.

Зміст лекції

22.1 Одновимірні масиви як параметри функції.

Масив як формальний параметр функцій можна оголосити за допомогою двох тотожних виразів:

```
тип ідентифікатор масиву[];
тип * ідентифікатор вказівника;
```

Наприклад:

```
func(int mas[], int n);
func(int *p, int n);
```

На прикладі однієї функції розглянемо різні варіанти її реалізації і передачі параметрів.

Нехай потрібно порахувати середнє арифметичне елементів масиву. Це можна зробити обома запропонованими способами:

```
double func1(double a[], int n){
    int i;
    double s;
    for(i=0, s=0; i<n;i++)s+=a[i];
    return s/n;
}
double func2(double *p, int n){
    double s=0, *q=p+n;
    for(;p<q;p++)s+=*p;
    return s/n;
}
```

Обидва варіанти вимагали крім передачі адреси масиву до функції також передати його розмір.

Приклад виклику обох варіантів із **main()**

```
int main()
{
  double mas[100], s1,s2, *ptr=&mas[10];
  int i;
     srand(time(NULL));
```

```
for(i=0; i<100; i++)mas[i]=rand()%1000*0.01;
     s1=func2(mas, 100);    printf("%f\n", s1);
     s2=func1(ptr,10); printf("%f\n", s2);
return 0;
}
```

Як видно з наведеного прикладу, виклик в обох випадках може бути однаковим.

22.2 Рядки як параметри функції

Те, що було сказане для масивів, можна віднести і до рядків. Відміна полягає у тому, що рядок обмежений нуль-символом, тому відпадає необхідність вказувати його розмір у списку параметрів функції.

Для роботи з рядком зручніше використовувати вказівники.

Завдяки тому, що до функції передано вказівник на масив, редагувати можна саме його вміст, а не його копії.

Нехай, наприклад, потрібно поміняти регистр букв у переданому як парамет рядку.

```
char* change(char*str)
{
    while(str[i]!='\0'){
         if(islower(str[i]))
              str[i]=toupper(str[i]);
         else
              if(isupper(str[i]))
                       str[i]=tolower(str[i]);
         i++;
         }
     return str;
}
int main()
{
  char str[100];   int i=0;
  gets(str);
  change(str);
  puts(str);
  return 0;
}
```

Наведений приклад демонструє загальний підхід до функцій по роботі з рядками: такий самий результат можна і повернути. Можна також повернути потрібну частину модифікованого рядка.

22.3 Двовимірні масиви як параметри функції

Двовимірний масив можна передати до функції як параметр декількома способами. Якщо розглядати його як масив одновимірних масивів, то синтаксис такого формального параметра наступний:

```
тип ідентифікатор масиву[][розмірність];
```

або

```
тип(*ідентифікатор масиву)[розмірність];
```

В обох випадках розмірність другого виміру вказувати обов'язково.

Якщо пам'ятати, що безпосередньо елементи двовимірного масиву розташовані у пам'яті підряд, можна звернутися за адресою першого елемента масиву. Але у цьому випадку з ним потрібно буде працювати як з одновимірним масивом і індекси вираховувати вручну. Синтаксис оголошення такого формального параметра наступний:

```
тип*ідентифікатор вказівника на нульовий елемент;
```

Приклади оголошення і виклику функцій з такими параметрами наведені у наступній програмі.

```
#define N 5
#define M 10

void gen(int n, int m, int a[][M]);
void print(int n, int m, int(*a)[M]);
void sort(int n, int m, int *p);
void swap(int*p, int*q){int t=*p; *p=*q; *q=t;};

int main(){
int arr[N][M], n=N, m=M;
 srand(time(NULL));
 gen(n, m, arr);
 print(n, m, arr);
 sort(n, m, &arr[0][0]); printf("\n");
 print(n, m, arr);
 return 0;
}

void gen(int n, int m, int a[][M]){
int i,j;
 for(i=0;i<n;i++)
      for(j=0;j<m;j++)
           *(*(a+i)+j)=rand()%100;
};

void print(int n, int m, int(* a)[M]){
int i,j;
```

```
 for(i=0;i<n;i++){
      for(j=0;j<m;j++)
           printf("%3d" ,(*(a+i))[j]);
      printf("\n");
 }
};

void sort(int n, int m, int *p){
 int nm=n*m, *q=p+nm-1, *r,*rm;
 for(;p<q;p++)
 {
      for(r=p+1, rm=p;r<q-1;r++)
           if(*r<*rm)rm=r;
      swap(p,rm);
 }
};
```

Можливий сеанс роботи

```
 67 67 65 49 56 87  8 15  7 82
 54 32 37 60 57 72 94  8 42 85
  9  3 18 12 13 53 29 98 97 24
 94 23 65 98 67 72 26 26 67 20
 89 20 40 28 78 12 42  9 31 54

  3  7  8  8  9  9 12 12 13 15
 18 20 20 23 24 26 26 28 29 32
 37 40 42 42 49 53 54 56 57 60
 65 65 67 67 67 67 72 72 78 82
 85 87 89 94 94 97 98 98 31 54
```

Зрозуміло, що ця програма коректно працюватиме тільки за умови **n=N, m=M.**

Контрольні запитання

1. Які варіанти оголошення одновимірного масиву як формального параметра функції ви знаєте?
2. Як виглядатимуть виклики відповідних функцій?
3. Які варіанти оголошення символьного рядка як формального параметра функції ви знаєте?
4. Як повернути новий символьний рядок із функції?
5. Які варіанти оголошення двовимірного масиву як формального параметра функції ви знаєте?
6. Як залежить від варіанту оголошення використання формального параметра – двовимірного масиву в тілі функції?

Лекція №23

Тема лекції: Класи пам'яті

План лекції

1. Час життя, область дії та область визначення
2. Автоматичні змінні
3. Клас пам'яті **`extern`**
4. Клас пам'яті **`static`**.
5. Клас пам'яті **`register`**
6. Передача параметрів за адресами

Зміст лекції

Відкомпільована С-програма створює і використовує чотири логічно розділених області пам'яті, що мають своє призначення.

Перша містить код програми, друга призначена для зберігання глобальних змінних. Третя – стек, який містить локальні змінні, через нього реалізована передача параметрів у функцію і повернення значень із неї. Його також використовують для зберігання поточного стану процесора. Четверта – динамічна пам'ять (купа) – це область вільної пам'яті, яку у програмі можна динамічно розподіляти. Схематично це можна представити так:

Оголошення змінної за синтаксисом має вигляд:

```
[клас пам'яті][const]тип ідентифікатор[ініціалізатор];
```

І із оголошення зрозуміло, глобальна ця змінна або локальна. В С(С++) змінна може належати одному з чотирьох класів пам'яті, а саме: **auto**, **extern**, **static**, **register**.

23.1 Час життя, область дії та область визначення

Клас пам'яті визначає час життя, область дії і область видимості змінної. Якщо він не заданий явно, то може бути визначений із контексту оголошення, оскільки саме воно задає її область дії, область видимості та час життя.

Область дії змінної залежить від опису і його розміщення у тексті програми. Це та частина програми, де змінну можна використовувати для доступу до зв'язаної з нею області пам'яті. Залежно від області дії змінна може бути локальною або глобальною.

Час життя може бути постійним – на час виконання усієї програми і тимчасовим – тільки на час виконання блоку.

Область видимості – та частина тексту програми, з якої є доступ до зв'язаної з нею області пам'яті. Як правило, співпадає з областю дії. Виняток – використання того самого ідентифікатора у внутрішньому блоці. Зовнішня змінна у внутрішньому блоці у такому випадку невидима, хоча цей блок і входить до її області дії Наприклад:

```
int count=10;
      printf("%d ", count);
      if(1)
      {     //інша змінна, але з таким самим ідентифікатором
            int count=20;
            printf("%d ", count);
      }
      printf("%d \n", count);
```

У розглянутому прикладі фрагменту функції С++ наведена ситуація, коли область дії змінної **count** і її область видимості не співпадають.

23.2 Автоматичні змінні

Специфікатор **auto** визначає локальну автоматичну змінну – пам'ять під неї виділена в стеку, тому її обов’язково потрібно ініціалізувати кожного разу при вході в блок, де вона визначена. При виході з блоку пам'ять кожного разу звільняється.

Оголошення локальної змінної може бути:

- у середині блоку (для C++);
- у середині функції (як окремий випадок блоку);
- як формальний параметр у заголовку функції.

Час життя – з моменту оголошення до кінця блоку. Приклад оголошення:

```
auto char str[80];
```

Специфікатор **auto** на внутрішньому рівні можна не використовувати – для локальної змінної це клас пам'яті за умовчанням.

Якщо функція має параметри, вони оголошуються у списку параметрів і діють в середині функції як локальні автоматичні змінні. Такі змінні будуть ініціалізовані значеннями, яких набудуть у момент виклику, наприклад, функція визначає чи входить даний символ до даного рядка:

```
int is_symb(char*s,char symb){
 while(*s){
      if(*s==symb) return 1;
      s++;
      };
 return 0;
}
```

Тут змінні **s** та **symb** –автоматичні змінні.

23.3 Клас пам'яті extern

Це глобальні змінні, їх оголошують на зовнішньому рівні і до них застосовується так зване зовнішнє зв’язування, тобто вони будуть доступні у всій програмі, навіть якщо вона складатиметься з декількох файлів.

Для будь-якої змінної визначені дві дії – оголошення та опис. Оголошення визначає тип та ім’я змінної, а опис виділяє під неї пам'ять. Тому змінна може бути оголошена декілька разів, а описана тільки один раз.

Як правило, оголошення й опис змінної збігаються, але для глобальної змінної **extern** це не обов’язково.

Якщо змінна визначена перед основною функцією програмного комплексу, то в інших функціях, що входять у програмний файл, її можна не описувати як зовнішню, вона й так діятиме в них. До них можна отримати доступ у будь-якому виразі, незалежно від того, в якій функції знаходиться даний вираз.

Опис змінної як **extern** усередині функції потрібний у тих випадках, коли цю змінну необхідно використати, а вона визначена або у функції, яка активізується пізніше, або в іншому програмному файлі.

Наприклад:

```
int count1=100, count2=200;

int main(){
extern int count1;
printf("count1=%d, count2=%d\n", count1, count2);
}
```

Тут змінні **count1** та **count2** – обидві глобальні і на екрані отримаємо

```
count1=100, count2=200
```

Те саме можна сказати і про оголошення на зовнішньому рівні в іншому файлі – всі оголошення підключеного файла стають доступними після виконання директиви **#include.** На цьому принципі основано використання функцій, прототипи яких оголошені у підключених заголовкових файлах а також деяких стандартних макрооголошень(**NULL, PI, MAX_INT**, тощо).

23.4 Клас пам'яті static

Специфікатор **static** відрізняється від **extern** областю видимості, а від **auto** – часом життя. Змінні можуть бути описані і на глобальному і на локальному рівнях.

Локальні статичні змінні оголошують на внутрішньому рівні, але компілятор виділить під них пам'ять як для глобальних. Тому область видимості у них буде локальна, а час життя – глобальний, тобто з моменту оголошення до закінчення роботи програми. Синтаксис:

```
static тип ідентифікатор[ініціалізатор];
```

Статичні змінні одразу ініціалізовані за умовчанням відповідно типу. Можна ініціалізувати іншими значеннями, наприклад:

```
static int x, y=10;
```

Використовують такі змінні у випадках, коли потрібне збереження значень між викликами функцій. Наприклад, нехай на екран потрібно періодично виводити чергову степінь двійки, тоді функція для генерації степенів може виглядати так

```
int power_2(){
      static int gen=1;
```

```
        gen *=2;
        return gen;
 }
```

А щоб продемонструвати коректність ii роботи, можна скористатись циклом у викликаючій функції:

```
int main(){ . . .  m=n;
for(;n!=0;n--) k=power_2();
printf("2^%d=%d\n", m, k);
 }
```

У наведеному прикладі ініціалізація статичної змінної відбувається явно.

Глобальні статичні змінні. Оголошення змінних зі специфікатором **static** на глобальному рівні примушує компілятор створити глобальні змінні, але з областю видимості тільки у тому файлі, де вони оголошені. Тобто така змінна глобальна, але недоступна для функцій із інших файлів. Скористатись статичними змінними із іншого файлу можна лише за допомогою функцій того самого файлу, де ці змінні оголошені**.**

Прикладом можуть слугувати функкції, яка задає значення статичної змінної **srand()** – значення першого псевдовипадкового числа та **rand(),** яка це значення використовує

23.5 Клас пам'яті register

Надає можливість найскорішого доступу до об’єкту завдяки тому, що змінні зберігаються безпосередньо в регистрах. Якщо регистри зайняті, то змінні будуть опрацьовані як **auto** без повідомлень .

Приклад використання обчислення степені **n** числа **a**:

```
int int_pwr(register int a, register int n)
{         register int temp=1;
for(;n;n--)temp*=a;
return temp;
}
```

Відчутний ефект від використання насправді тільки для **int** та **char**.

23.6 Передача параметрів за адресами

Передача параметрів до функції приводить до створення локальних автоматичних змінних, що діють усередині функції, фактично це копії тих змінних, що були визначені перед викликом функції і передані в неї як параметри. Тому зміни, що відбулися з цими змінними у функції – це зміни

копій, а не самих змінних, що були використані як фактичні параметри в момент виклику функції.

Виключення – змінна, значення якої функція повертає.

Якщо потрібно, щоби функція працювала безпосередньо із змінними із викликаючої функції, потрібно передавати не їх значення, а їх адреси.

Класичний приклад – поміняти місцями значення двох змінних. Наприклад:

```
void swap(double* a, double* b){double t=*a; *a=*b; *b=t;}
int main()
{
   double a, b;
      printf("enter a and b\n"); scanf("%lflf",&a,&b);
      swap(&a, &b);
      printf("a=%f, b=%f\n", b, a);. . .
}
```

Контрольні завдання та запитання

1. Що ви розумієте під терміном *час життя*?
2. Чим відрізняються поняття *область дії* та *область видимості*?
3. У яких випадках обов'язково потрібно використовувати ключове слово **`auto`** в оголошенні змінної?
4. Як компілятор реагуватиме на ключове слово **`register`**?
5. Які варіанти передачі параметрів у функцію ви знаєте?
6. Чим відрізняється використання змінних, що належать до класів пам'яті **`extern`** та **`static`**?

Лекція №24

Тема лекції: Вказівники на функції

План лекції

1. Оголошення вказівника на функцію
2. Звертання через вказівник на функцію
3. Вказівник на функцію як параметр функції
4. Деякі приклади з використанням вказівників на функцію
5. Функції з невизначеною кількістю параметрів

Зміст лекції

24.1 Оголошення вказівника на функцію

Ідентифікатор функції є константним вказівником на перший байт виконуваного коду функції і визначає **точку входу** у функцію. Коли процесор передає управління на функцію, то фактично виконання програми буде продовжено з даної адреси.

Так само як і для масиву, можна визначити вказівник на функцію, який уже не буде константним і з яким можна буде працювати. Синтаксис оголошення такого вказівника відрізняється від оголошення вказівників на змінні і виглядає так:

```
тип(*ідент.вказ.)([список типів параметрів]);
```

Вказівник на функцію вводиться окремо від оголошення та визначення будь-якої функції і визначає, до якої саме групи функцій може бути застосований, а саме: тільки до функцій із заданою схемою формальних параметрів і типом результату. Наприклад оголошення

```
double (*pf)(int, double*);
```

визначає, що змінна **pf** може бути використана як вказівник на будь-яку функцію, що повертає **double** і має два параметри – **int** та **double***. Порядок розташування параметрів має значення.

24.2 Звертання через вказівник на функцію

Вказівнику на функцію може бути присвоєний інший вказівник на функцію зі схожим прототипом. Наприклад:

```
void (*pf)(void);  // вказівник на функцію без  параметрів
void (*qf)(void);
qf=pf;
```

Вказівнику на функцію можна присвоїти значення відповідної функції:

```
ідент.вказівника=ідент.функції;   або
ідент.вказівника=&ідент.функції;
```

Наприклад:

```
pf=sum;
pf=&sum;
```

Вказівник на функцію – це змінна, що містить адресу деякої функції, тому може бути використаний для виклику цієї функції. Синтаксис виклику функції за допомогою розадресації вказівника (перший варіант):

```
(*ідент.вказівника)(список фактичних параметрів);
```

Такий варіант виклику явно вказує на застосування саме вказівника на функцію, а не просто функцію.

У сучасних компіляторах допустимий також інший різновид виклику, схожий на виклик звичайної функції, (другий варіант):

```
ідент.вказівника(список фактичних параметрів);
```

Але краще дотримуватись першого варіанту виклику, щоб код програми був зрозуміліший для людини.

Наприклад:

```
void print(int n){ /* реалізація          */   };
. . .
void (*qf)(int);     // оголошення вказівника в main()
qf=print;        // тотожно qf=&print;
(*qf)(10);             // перший варіант з розадресованим вказівником
qf(10);                // другий варіант виклику
```

Арифметичні операції над вказівниками на функції заборонені.

24.3 Вказівник на функцію як параметр функції

Вказівники на функції часто використовують як параметр іншої функції. Замість того, щоб розробляти декілька різних великих функцій, відміни яких – це незначні деталі, які можуть бути оформлені як окремі декілька функцій,

прототипи яких відрізнятимуться лише ідентифікатором, можна цю відмінність передати як параметр функції.

Наприклад, потрібно написати функцію визначення максимального або мінімального елемента масиву, у яку як параметри передаються: сам масив, його розмірність і вказівник на функцію для порівняння.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#define n 10
double less(double a, double b){
     if(a<b) return a;return b;
}
double grt(double a, double b){
     if(a>b) return a;return b;
}
void gen(int n, double a[]){
int i=0;
     for(;i<n;i++)a[i]=0.01*(rand()%100);
}
double extrem(int n, double *a, double(*pf)(double, double)){
double max=a[0];
int i=1;
    for(; i<n; i++) max=(*pf)(max,a[i]);
    return max;
}
void out(int n, double*p){
    while(n-1){
          printf("%5.2f ",*p);
          p++;n--;
    };
    printf("\n");
};
int main(){
    srand(time(0));
    double a[10], m;
    gen(n,a);
    out(n,a);
    m=extrem(n, a, less);
    printf("min=%lf\n",m);
    m=extrem(n, a, grt);
    printf("max=%lf\n",m);
return 0;
}
```

У наведеному прикладі функція **extrem** викликана двічі – один раз з підстановкою функції **less** у якості фактичного параметра, другий раз – функції **grt.**

24.4 Деякі приклади з використанням вказівників на функцію

Деякі обчислювальні алгоритми зручно реалізувати як окремі функції. Це алгоритми, відомі зі шкільного курсу математики. Вони цікаві тим, що можуть бути розглянуті як приклад використання вказівників на функції в якості параметрів іншої функції.

Наприклад, пошук кореня рівняння, яке складно розв'язати математично, методом ділення відрізка навпіл. Це так званий *метод дихотомії*.

Постановка задачі. Відомий відрізок, де може знаходитись тільки один з коренів заданого рівняння **[a,b].** Знайти цей корінь з точністю **ε**. Рівняння задане у вигляді **f(x)=0.**

Ідея методу. Відрізок ділимо навпіл і перевіряємо, чи можна вважати отриману точку **c** коренем рівняння з точністю **ε**, тобто що **|f(c)|<ε.** Якщо так, потрібний корінь з наданою точністю знайдено, якщо ні – обираємо новий відрізок, де знаходиться шуканий корінь – **[a, c]** або **[c, b],** а саме той з них, на граничних точках якого функція **f** матиме різні знаки і переходимо до нової ітерації.

Реалізація такої функції може мати вигляд:

```
double root(double a, double b, double eps,
                                    double(*f)(double)){
  double c, ff;
  char ch;
  a=a<b?a:b;
         do{
              c=(b-a)/2+a;
              ff=(*f)(c);
              if((*f)(a)*ff>0)a=c; else b=c;
         }while(fabs(ff)>=eps);
 return c;
}
```

Фактичним параметром у викликах такої функції може бути будь-яка стандартна або користувацька функція з відповідним прототипом.

24.5 Функції зі змінною кількістю параметрів

Список формальних параметрів може закінчуватися трьома крапками (...), що означає, що число аргументів функції змінне: на початку списку – декілька обов'язкових параметрів, після яких довільна кількість необов'язкових.

Синтаксис оголошення наступний:

```
тип ident.функції(сп.обов'язкових параметрів,...)
```

Виклик такої функції повинен містити принаймні стільки обов'язкових аргументів, скільки формальних параметрів задано перед останньою комою у списку параметрів. Під час виклику компілятор перевірятиме узгодження типів тільки для обов'язкових параметрів, оскільки для необов'язкових така інформація відсутня.

Прикладами таких функцій можуть бути функції **`printf()`** і **`scanf()`.**

За перевірку типів і організацію коректної роботи з параметрами такої функції повністю відповідає програміст, кількість параметрів передають до функції через один із обов'язкових аргументів.

Словник. Точка входу, вказівник на функцію, тип функції, функція зі змінною кількістю параметрів, обов'язкові і необов'язкові параметри.

Завдання на СРС:

1)Масиви вказівників на функцію[6, с.137]

2) Функції, що повертають вказівник на функцію[2,с.77, 3, с.138]

Контрольні запитання

1. Що таке точка входу функції?
2. Як оголосити вказівник на функцію? Які значення можна йому присвоїти?
3. Які правила оголошення функції зі змінною кількістю параметрів?

Лекція №25

Тема лекції: Структури

План лекції

1. Оголошення та ініціалізація.
2. Розмір структури.
3. Звертання до елементів структури.
4. Структури як поля структури.
5. Масиви як поля структур і масиви структур.

Зміст лекції

25.1 Оголошення та ініціалізація.

Структура – це складений об'єкт, до складу якого входять елементи будь-яких типів, за винятком функцій. На відміну від масиву, який складається з однотипних елементів, до складу структури можуть входити елементи як однакових, так і різних типів.

Таким чином, структура – це тип даних, сформований з об'єктів, що належать до одного або різних типів даних.

Допустимо окремо визначити тип і об’єкт (змінну) цього типу, допустимо це зробити одночасно. Загалом допустимі наступні варіанти.

Попереднє оголошення іменованого типу. Синтаксис:

```
struct ідентифікатор_типу {список_компонентів};
```

Наприклад:

```
struct Person{
    char name[30];
    int day, month, year;
    };
```

Оголошення змінних структурного типу в цьому випадку нічим не відрізняється від оголошень змінних інших типів:

```
struct ідентифікатор_типу список_ідентифікаторів_змінних;
```

Для розглянутого прикладу:

```
struct Person a1, a2, *house, group[20];
```

Змінні структурного типу, як і будь-якого іншого, можна одразу ініціалізувати, наприклад:

```
struct Person fedir={"Fedir Fedorenko",2,5,1992};
```

Суміщення оголошення структурного типу і структурного об'єкту.

Синтаксис:

```
struct [ідентифікатор_типу] {
                    список_компонентів
     }список_ідентифікаторів_змінних;
```

Приклад:

```
struct Person{
     char name[30];
     int  year;
     } a, b[10], fedir={"Fedir Fedorenko",1992};
```

Якщо інших структурних об'єктів такого типу не заплановано використовувати у даній області видимості, то власну назву типу можна опустити, наприклад:

```
struct {
     char name[30];
     int  year;
     } a, b[10], fedir={"Fedir Fedorenko",1992};
```

У межах одної програми допустима тільки одна неіменована структура.

Якщо об'єкт оголошений на зовнішньому рівні і не ініціалізований явно, він буде ініціалізований за умовчанням. Усі поля приймуть нульові значення відповідного типу.

Значення полів об'єкту, ініціалізованого на внутрішньому рівні, не визначені.

Ініціалізація таких об'єктів може відбуватися одночасно з оголошенням, як показано на попередньому прикладі. Якщо такий об'єкт не ініціалізований одразу, то після оголошення значення полям надають окремо кожному.

25.2 Розмір структури.

Для визначення розміру пам'яті необхідно скористатись оператором

```
sizeof (ідентифікатор_об'єкту) або
sizeof (ідентифікатор_типу)
```

Отримане значення може бути більше, ніж арифметична сума розмірів пам'яті змінних-полів структури.

25.3 Звертання до елементів структури

В якості полів структури можна використовувати об’єкти будь-яких типів, окрім функцій. Це можуть бути змінні вбудованих типів, інших структурованих типів (структур, масивів, рядків, об’єднань, перерахувань, тощо) та вказівники на всі ці типи.

Для доступу до поля структурного об’єкту використовують оператор **крапка – ’.’**

ідентікатор_ змінної.ідентифікатор_ поля

Наприклад:

```
struct Point{
     int x, y, z;
     } a; //  ініціалізація на зовнішньому рівні
int main(){
printf(" x=%d, y=%d, z=%d\n",a.x, a.y, a.z);
. . . }
```

Якщо доступ потрібно організувати через вказівник на такий об’єкт, то потрібно використовувати оператор **стрілочка – "->".**

вказівник_на_структуру->ідентифікатор_поля

Наприклад:

```
struct Point{
     int x, y, z;
     } *p; . . .
printf("x=%d, y=%d, z=%d\n",a->x, a->y, a->z);
```

Потрібно звернути увагу, що вказівник може бути розіменований і тоді доступ – прямий, через крапку. Для останнього прикладу:

```
printf("x=%d, y=%d, z=%d\n",(*p).x,(*p).y,(*p).z);
```

Оператор **присвоювання =** для змінних структурного типу визначений **,** але працює лише для однотипних змінних і виконує побайтне присвоєння значень відповідних полів, наприклад:

```
struct Point{
     int x, y, z;
     } ob1={0,10,20}, ob2;
ob2=ob1;
```

Після оголошення будь-якого структурного типу, його можна використовувати як інші типи, тобто можна оголосити змінну такого типу,

вказівник на таку змінну, масив таких змінних, передати як параметр у функцію, виділити під неї динамічну пам'ять.

25.4 Структури як поля структури

Структура може містити іншу іменовану або неіменовану структуру як внутрішню, оголошену одночасно з оголошенням зовнішньої структури прямо у списку компонентів.

```
struct [ідентифікатор типу1] {
     список компонентів1;
     struct [ідентифікатор типу2] {
          список компонентів2;
     }[список ідентифікаторів змінних2];
 }[список ідентифікаторів змінних1];
```

Наприклад:

```
struct Person{
     char name[30];
     struct {
          int day, month, year;
     }birthday;
} obj;
```

Звернення до поля такої структури:

```
printf("my birthday =%d:%d:%d \n", obj.birthday.day,
          obj.birthday. month, obj.birthday. year);
```

Внутрішніх іменованих структур може бути декілька, вони можуть бути вкладеними.

Полем структури може бути змінна структурного типу. Оголошеного окремо. Використання таких полів не відрізняється від полів інших типів. Наприклад:

```
struct date{
     int day, month, year;
};
struct Person{
     char name[30];
     struct date birthday;
} obj;
```

Звернення до поля такої структури:

```
printf("my birthday =%d:%d:%d \n", obj.birthday.day,
          obj.birthday. month, obj.birthday. year);
```

Як видно із розглянутих прикладів, звернення до поля структури в обох випадках відбудеться однаково – через його ідентифікатор.

25.5 Масиви як поля структур і масиви структур

Якщо масив – поле структури, звернення до його елемента може здійснюватись як за допомогою індексного виразу, так і з використанням вказівника. Наприклад:

```
struct Polygon{
     int n;
     double coord[3];
  }a, *pb;
```

Доступ до елементів масиву:

```
a.coord[1]= pb->coord[1];
```

Якщо оголошений масив елементів структурного типу, робота з його елементами нічим не відрізнятиметься від роботи з масивами інших вбудованих або користувацьких типів. Наприклад:

```
struct Person{
     char name[30];
     struct date birthday;
 } arr[5], *p=arr;
```

Доступ до елементів масиву:

```
if(arr[0].birthday.year>arr[1].birthday.year)
          printf("student %s elder\n",arr[0].name);
```

Передача елементів структур у функцію нічим не відрізняється від передачі у функцію простих змінних, окрім наявності складеного імені змінних як у наведеному вище прикладі з викликом функції **printf.**

Передача структурних змінних як параметрів у функцію і повернення відповідних значень, як і для простих змінних, може відбуватись і за значенням, і за вказівником. Наприклад:

```
int older(struct Person arr[], int n);
```

Попередньо їх потрібно оголосити з використанням іменованого типу.

Словник. Структурний тип, структурна змінна, структура, поле структури, тип користувача, операція **крапка**, операція **стрілочка.**

Контрольні запитання

1. Які існують варіанти оголошення змінної структурного типу?

2. У яких випадках застосовують оператор точка, а у яких – стрілочка?
3. Які є обмеження на типи полів структури?
4. Як ініціалізують поля структури?
5. Як визначити розмір пам'яті, яку займає змінна структурного типу?
6. Як можна звернутися до поля структури, яка є полем іншої?
7. Як звернутися до поля елемента структури у масиві структур?
8. Як звернутися до елемента масиву, який є полем структури?

Лекція №26

Тема лекції: Структури. Робота зі змінними структурного типу

План лекції

1. Перейменування типів
2. Вказівники на структури і вказівники як поля структури.
3. Деякі алгоритми роботи із структурами

Зміст лекції

Визначення структурного типу дає можливість уводити новий, користувацький тип.

26.1 Перейменування типів

Мова С дозволяє визначати імена нових типів даних за допомогою ключового слова **typedef**. Насправді таким способом новий тип даних не створюється, а всього лише визначається нове ім'я для вже існуючого типу.

Стандартний вид оператора **typedef** наступний:

```
typedef тип ідентифікатор;
```

Тут **тип** – це будь-який існуючий тип даних, а **ідентифікатор** – це нове ім'я для даного типу. Нове ім'я визначається на додаток до існуючого імені типу, а не заміщає його. Наприклад:

```
typedef float real;
```

Даний оператор повідомляє компілятору про необхідність розпізнавання **real** як іншого імені для **float**. Тепер можна оголосити змінну **var** з використанням типу **real**:

```
real var;
```

Так само можна створити синонім будь-якого структурного типу, що значно спростить оголошення майбутніх змінних. Синтаксис:

```
typedef struct [ідентифікатор_типу] {
          список компонентів
     }альтернативний_ідентифікатор_типу;
```

Наприклад:

```
typedef struct human{
          char name[30];
          int year;
     }PERSON;
     . . .
```

```
PERSON someperson, arr[10];
```

Як видно з оголошення, **ідентифікатор_типу** може бути відсутній

26.2 Вказівники на структури і вказівники як поля структури

Приклади з вказівниками на структури були розглянуті раніше.

Полями структури можуть бути вказівники на інші типи, у тому числі і структуровані. Приклад оголошення поля – вказівника під майбутнє виділення динамічної пам'яті під масив.

```
struct Array{
        int n;
        double *p;
 }a1, a2;
```

Доступ до елементів масиву:

```
printf("enter size");
 scanf(" %d", a1.n);
a1.p=(double*)calloc(a1.n,sizeof(double));
 for(i=0; i< a1.n; i++){
      scanf(" %lf", &a1.p[i]);
}
```

У розглянутому прикладі оголошено тип – динамічний масив дійсних чисел **struct Array** і змінні такого типу **a1** та **a2.** Для змінної **a1** під масив **p** виділена пам'ять розміру **n** і з клавіатури заповнені всі елементи масиву.

Для полів структури допустимим є посилання на об'єкт такого самого типу. Наприклад, структура, яка описує вузол однозв'язного списку, виглядатиме так:

```
 struct Node{
      char inf[10];
      struct Node*next;
 }*head;
```

Тут другим полем структури є вказівник на неї.

26.3 Деякі алгоритми роботи із структурами

Можна розглянути як приклад побудову однозв'язного списку додаванням чергового елементу в початок і виведенням на екран вмісту побудованого списку.

```
struct Node{
     char inf[10];
     struct Node*next;
 }*head;
```

```
struct Node*create();
struct Node*add_first(struct Node* h);
int main()
{
     int i, n;
     struct Node *cur;
  printf("How many nodes?\n"); scanf(" %d", &n);
  getchar();
  if(n)
  {
     head=create();
     for(i=1; i<n; i++)     head=add_first(head);
     for(cur=head;cur!=NULL;cur=cur->next)
          printf("%s ",cur->inf);
     }
return 0;
}
struct Node*create()
{
     struct Node* temp=
          (struct Node*)malloc(sizeof(struct Node));
     temp->next=NULL;
     puts("enter sent");
     gets(temp->inf);
     return temp;
}
struct Node*add_first(struct Node* h)
{
     struct Node* temp=
          (struct Node*)malloc(sizeof(struct Node));
     temp->next=h;
     puts("enter sent");
     gets(temp->inf);
          return temp;
}
```

Словник. Перейменування типу.

Контрольні запитання

1. Як можна визначити користувацький тип в С?
2. Чи можна визначити новий тип за допомогою засобу **typedef**?
3. У яких випадках можна використовувати вказівник як поле структури?
4. Чи може структура містити вказівник на себе?

Лекція №27

Тема лекції: Об’єднання. Бітові поля

План лекції

1. Оголошення та ініціалізація
2. Розмір об’єднання
3. Звертання до елементів об’єднання.
4. Об’єднання як поле структури і навпаки
5. Бітові поля

Зміст лекції

27.1 Оголошення та ініціалізація

Об’єднання – спеціальний тип даних, який дає можливість записувати до одної ділянки пам'яті змінні різних типів. Оголошення виглядає так само як у структури, як і у структури можна тип оголошувати окремо або разом із змінними, можна користуватися **typedef,** різниця у розташуванні полів у пам'яті.

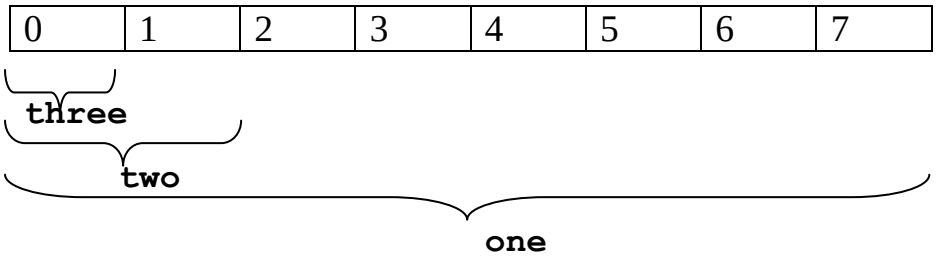
Синтаксис оголошення:

```
union ідентифікатор_типу{список_компонентів};
```

Наприклад:

```
union Uni  {
       double one;
       int two;
       char three;
  }
```

Тільки розташовані ці поля будуть не одне за одним, як у структурі, а одне НА одному, тобто одночасно можна звернутися тільки до одного поля. Фактично, інформація там буде однакова. Просто звернутися до неї користувач може або як до цілого, або як до дійсного, або як до символу.



На малюнку видно, що накладання значень полів відбувається зі сторони молодших байтів.

27.2 Розмір об'єднання

Розмір пам'яті, виділений під об'єднання, відповідатиме розміру найбільшого поля, тобто у наведеному прикладі

sizeof(union Uni)= sizeof(three).

У загальному випадку можна сказати, що

sizeof(ідентифікатор_об'єднання)≥ max(список_компонентів)

Ініціалізувати таку змінну можна тільки значенням, сумісним з типом першого поля. Наприклад, **union Uni high{-35.05}**

27.3 Звертання до елементів об'єднання

До змінних з типом об’єднання можна застосовувати всі операції, що й до структурних змінних: присвоювання (тобто копіювання цілого об’єднання), визначення адреси змінної, визначення розміру пам'яті, виділення окремих елементів (полів) об’єднання. Для звертання до полів об’єднання використовують такі ж синтаксичні конструкції на основі операцій «крапка» і «стрілка», як і в разі звертання до елементів структур:

```
ідентифікатор_змінної_об’єднання.ідентифікатор_поля
вказівник_на_об’єднання-> ідентифікатор_поля
```

27.4 Об’єднання як поле структури і навпаки

Поля об’єднань можуть бути довільного типу – простого чи складеного. Зокрема, поле об’єднання може бути структурою, вкладеним об’єднанням або вказівником на структуру чи об’єднання. Водночас об’єднання можуть бути елементами масиву або складовими частинами структур інших типів.

Приклад використання об’єднання як поля структури:

```
union payment
{
      char card[25];
       long check;
};
enum paytype { CARD, CHECK };
struct pmnt
{
      paytype ptype;
      union payment pay;
};

int main()
```

```
{
 struct pmnt p;
 p.ptype = CHECK;
 p. pay.check = 123456;
 switch (p.ptype)
 {
      case CARD: printf(" %s\n", p. pay.card); break;
      case CHECK: printf(" %ld\n", p. pay.check);break;
      }
 return 0;
}
```

У наведеному прикладі разом з об’єднаннями зберігається інформація про те, який саме елемент використовується в даний момент, адже у кожний момент часу в змінній типу об’єднання зберігається тільки одне значення.

27.5 Бітові поля

Структури і об’єднання можуть містити члени, які займають менше простору в пам'яті, ніж цілочисельний тип. Ці члени визначені як бітові поля. Синтаксис оголошення бітового поля наступний:

```
[декларатор] : константний_вираз;
```

Тут **декларатор** – це ідентифікатор, за яким цей член буде доступним у програмі, обов’язково одного з цілих типів. **Константний_вираз** вказує кількість бітів, які член займає в структурі. Анонімні бітові поля можна використовувати для заповнення.

Приклад оголошення:

```
struct Date
{
        unsigned short nWeekDay  : 3;    // 0..7   (3 bits)
        unsigned short nMonthDay : 6;    // 0..31  (6 bits)
        unsigned short nMonth    : 5;    // 0..12  (5 bits)
        unsigned short nYear     : 8;    // 0..100 (8 bits)
};
```

Бітові поля не можуть існувати окремо, тільки як члени структури або об’єднання.

До бітового поля не можна застосувати операцію взяття адреси. Бітове поле можна ініціалізувати тільки константним виразом.

Словник. Об’єднання, бітові поля.

Контрольні запитання

1. Чим об’єднання відрізняється від структури?

2. Як підрахувати пам'ять, яка буде виділена під змінну даного об’єднання?
3. Як можна ініціалізувати об’єднання?
4. Як звернутися до поля об’єднання?
5. Чи може об’єднання бути полем структури? Чи може структура бути полем об’єднання?
6. У яких випадках використовують бітові поля?
7. Чи можуть бітові поля утворити користувацький тип?
8. Що є найменшою одиницею інформації, до якої можна звернутися за допомогою бітових полів?

Список використаної літератури

1. Т.В.Ковалюк Основи програмування К.: BVH, 2005. – 384с.
2. Вінник В.Ю. Алгоритмічні мови та основи програмування: мова С. – Житомир. Видавництво ЖДТУ. 2007. – 328с. [Електронне навчальне видання] http://programming.in.ua/programming/c-language.html
3. Шилдт, Герберт. Полный справочник по C, 4-е издание. : Пер. с англ. – М.: Издательский дом «Вильямс», 2002. – 704 с
4. Уэйт М., Прата С., Мартин Д. Язык Си. Руководство для начинающих. – М.:Мир,1988 – 512с.
5. Керниган Б., Ритчи Д. Язык программирования Си. – М.: Финансы и статистика, 1992. – 272с.
6. Т.А.Павловская Программирование на языке высокого уровня СПб.:Питер, 2005. – 461с.
7. С.А. Абрамов, Г.Г.Гнездилова, Е.Н. Капустина, М.И. Селюн. Задачи по программированию. – М.: Наука, 1988. – 224с. (рус.)
8. І.В.Караюз, І.В.Назарчук, О.Л.Тимощук Алгоритмізація та основи програмування: Методичні вказівки до комп’ютерного практикуму – К.: НТУУ «КПІ» 2017 – 92с.[Електронне навчальне видання]
9. Войтенко В.В., Морозов А.В. С та С++. Теорія та практика. – Житомир. Видавництво ЖДТУ. 2004. – 324с.
10. Мейлахс А.Л. Практикум по математическим основам информатики. Часть 1. Системы исчисления. Двоичная арифметика. Представление чисел в памяти ЭВМ -М., 2012 -63 с

Інформаційні ресурси

1. Електронний кампус НТУУ «КПІ ім. І. Сікорського» [сайт] / Єдине інформаційне середовище НТУУ «КПІ ім. І. Сікорського», 2011-2017. – Режим доступу: http://campus.kpi.ua
2. Бібліотека MSDN [Електронний ресурс]//Microsoft: [сайт]/ MSDN Library Numerical, 2012. – Режим доступу: http://msdn.microsoft.com/library/default.aspx

Додаток А
Типи С

Основні типи даних					
Тип	Позначення		Назва	Мін розмір пам’яти, байт (біт)	Діапазон значень
	Ім’я типа	Інші назви			
Цілий	int	signed signed int	Цілий	4 (32)	-2 147 483 648 до 2 147 483 647
	unsigned int	unsigned	цілий без знаку	4 (32)	0 до 4 294 967 295
	short	short int signed short int	короткий цілий співпадає з int на 16-розрядних	2 (16)	-32 768 до 32 767
	unsigned short	unsigned short int	короткий цілий без знаку	2 (16)	0 до 65 535
	long	long int signed long int	довгий цілий співпадає з int на 32-розрядних	4 (32)	-2 147 483 648 до 2 147 483 647
	unsigned long	unsigned long int	довгий цілий без знаку	4 (32)	0 до 4 294 967 295
	long long (C99)	long long int signed long long int	дуже довгий цілий	8 (64)	$-(2^{63}-1)$ до $(2^{63}-1)$
	unsigned long long (C99)	unsigned long long int	дуже довгий цілий без знаку	8 (64)	0 до 18 446 744 073 709 551 615($2^{64}-1$)
Символьний	char	signed char	байт (цілий довжина не менше 8 біт)	1 (8)	-128 до 127
	-	unsigned char	байт без знаку	1 (8)	0 до 255
	wchar_t (C++)	-	*розширенний символьний*	*2 (16)*	*0 до 65 535*
Дійсний	float	-	дійсний одинарної точності	4 (32)	1E-37 до 1E+37 (точність не менше 6 значущих цифр)
	double	-	дійсний подвійної точності	8 (64)	1E-37 до 1E+37 (точність не менше 10 значущих цифр)
	long double	-	дійсний максимальної точності	8 (64)	1E-37 до 1E+37 (точність не менше 10 значущих цифр)
	bool(C++)	-	*логичний*	*1 (8)*	*true (1)або false (0)*

Додаток Б
Основні функції із заголовкового файлу <math.h>

Назва	Опис
acos	арккосинус
asin	арксинус
atan	арктангенс
atan2	арктангенс з двома параметрами
ceil	округлення до найближчого більшого цілого числа
cos	косинус
cosh	гіперболічний косинус
exp	обчислення експоненти
fabs	абсолютна величина (числа з плаваючою точкою)
floor	округлення до найближчого меншого цілого числа
fmod	обчислення залишку від ділення без остачі для чисел з плаваючою точкою
frexp	розбиває число з плаваючою точкою на мантиссу і показник ступеня.
ldexp	множення числа з плаваючою точкою на цілу ступінь двох
log	натуральний логарифм
log10	логарифм за основою 10
modf	(x, p) витягує цілу і дробову частини (з урахуванням знака) з числа з плаваючою точкою
pow(x, y)	результат зведення x в ступінь y, xy
sin	синус
sinh	гіперболічний синус
sqrt	квадратний корінь
tan	тангенс
tanh	гіперболічний тангенс
Функції стандарту C99 (додатково)	
acosh	гіперболічний арккосинус
asinh	гіперболічний арксинус
atanh	гіперболічний арктангенс
cbrt	кубічний корінь
copysign (x,y)	повертає величину, абсолютне значення якої дорівнює x, але знак якої відповідає знаку y
erf	функція помилок
erfc	додаткова функція помилок
exp2 (x)	значення числа 2, зведеного в ступінь x, 2x
expm1 (x)	значення функції ex – 1
fdim(x, y)	обчислення позитивної різниці між x і y, fmax (x-y, 0)
fma(x,y,z)	значення функції (x * y) + z (див. FMA)
fmax(x, y)	найбільше значення серед x і y
fmin(x, y)	найменше значення серед x і y
hypot(x,y)	гіпотенуза, sqrt ($x^2 + y^2$)
Ilogb	експонента числа з плаваючою точкою, конвертована в int
Lgamma	натуральний логарифм абсолютного значення гамма-функці
Llrint	округлення до найближчого цілого (повертає long long)
lrint	округлення до найближчого цілого (повертає long)
llround	округлення до найближчого цілого в напрямку від нуля (повертає

	long long)
lround	округлення до найближчого цілого в напрямку від нуля (повертає long)
log1p (x)	натуральний логарифм 1 + x
log2	логарифм за основою 2
logb	ціла частина логарифма x за основою 2
nan(s)	Повертає нечислове значення 'Not a Number'
nearbyint	Округлення аргументу до цілого значення у форматі числа с плаваючою крапкою
nextafter(x,y)	Наступне найближче, що м.б. представлене для x (по напряму до y)
nexttoward(x,y)	Те саме, що й nextafter, але у має тип long double
remainder(x,y)	обчислює остачу від ділення відповідно стандарту IEC 60559
rint	округлення до цілого (повертає int) з викликом помилки inexact, якщо результат відмінний від аргументу.
remquo(x,y,p)	Те саме, що й remainder, але зберігає коефіцієнт по вказівнику p (як int)
round	Округлення до цілого (повертає int)
tgamma	гама-функція
trunc	округлення до найближчого цілого числа в напрямку до нуля

Додаток В

Операції мови С у порядку спадання пріоритету

Пріоритет	Знак	Семантика	Тип	Асоціативність
1	( )	Виклик функції	первинні	->
	[ ]	Звертання за індексом		
	.	Зверт до поля структ за знач		
	->	Зверт до поля структ за вказівн		
2	++	Постфіксний інкремент		
	--	Постфіксний декремент		
3	-	Зміна знака	унарні	<-
	++	Префіксний інкремент		
	--	Префіксний декремент		
	!	Логічне заперечення		
	~	Побітове заперечення		
	&	Визначення адреси		
	*	Звертання через вказівник (адресу)		
	(тип)	Перетворення до заданого типу		
	sizeof	Визначення розміру в байтах		
4	*	Множення	Арифметичні	->
	/	Ділення		
	%	Остача від ділення		
5	+	Додавання		
	-	Віднімання		
6	<<	Вліво	Зсуви побітові	
	>>	Вправо		
7	<		порівняння	
	<=			
	>			
	>=			
8	==			
	!=			
9	&	AND	порозрядні	
10	^	XOR		
11	|	OR		
12	&&	Логічне і	логічні	
13	||	Логічне або		
14	?:	Умовний	умовний	<-
15	=	Присвоєння		<-
	*=	Множення з присвоєнням	Присвоєння	
	/=	Ділення з присвоєнням		
	%=	Остача від ділення з присвоєнням		
	+=	Додавання з присвоєнням		

	-=	Віднімання з присвоєнням	Присвоєння	
	>>=	Зсув з присвоєнням		
	<<=	Зсув з присвоєнням		
	&=	Побітове and з присвоєнням		
	^=	Побітове xor з присвоєнням		
	|=	Побітове or з присвоєнням		
16	,	Послідовного обчислення	кома	->