Курс лекцій

«Алгоритмічне програмування»

В данному курсі представлено 9 лекцій (Введення вважається нульовою лекцією)

Вступ в алгоритмічне програмування

Основні поняття і визначення

Програми і програмне забезпечення

Визначення (ДСТ)
Програма - це дані, призначені для керування конкретними компонентами системи обробки інформації (СОІ) з метою реалізації визначеного алгоритму.

Визначення даються по: ДСТ 19781-90. Забезпечення систем обробки інформації програмне. Терміни і визначення. - М.:Изд-во стандартів, 1990.

Звернути увагу: програма - це дані. Один з основних принципів машини фон Неймана - те, що і програми, і дані зберігаються в одній і тій же пам'яті. Програма, яка зберігається в пам'яті, являє собою деякі коды, що можуть розглядатися як дані. Можливо, з погляду програміста програма - активний компонент, вона виконує деякі дії. Але з погляду процесора команди програми - це дані, які читає процесор й інтерпретує. З іншого погляду програма - це дані з точки зору обслуговуючих програм, наприклад, з погляду компілятора, який на вході одержує одні дані - програму на мові високого рівня (ЯВР), а на виході видає інші дані - програму в машинних кодах.

Визначення (ДСТ)
Програмне забезпечення (ПЗ) - сукупність програм СОІ і програмних документів, необхідних для їхньої експлуатації

Істотно, що ПЗ - це програми, призначені для багаторазового використання і застосування різними користувачами. У зв'язку з цим варто звернути увагу на ряд необхідних властивостей ПЗ.

1. Необхідність документування. По визначенню програми стають ПЗ тільки за наявності документації. Кінцевий користувач не може працювати, не маючи документації. Документація уможливлює тиражування ПЗ і продаж його без його розроблювача. По Бруксу помилкою в ПХ є ситуація, коли програмний виріб функціонує не у відповідності зі своїм описом, отже, помилка в документації також є помилкою в програмному виробі.
2. Ефективність. ПО, розраховане на багаторазове використання (наприклад, ОС, текстовий редактор і т.п.) пишеться і налагоджується один раз, а виконується багаторазово. Таким чином, вигідно переносити витрати на етап виробництва ПО і звільняти від витрат етап виконання, щоб уникнути витрати на тиражування.
3. Надійність. У тому числі:
    - Тестування програми при всіх допустимих специфікаціях вхідних даних
    - Захист від неправильних дій користувача
    - Захист від злому - користувачі повинні мати можливість взаємодії з ПЗ тільки через легальні інтерфейси.

Готьє: "Помилки в системі можливі через збої апаратури, помилок ПЗ, неправильних дій користувача. Перші - неминучі, другі - ймовірні, треті - гарантовані".
Поява помилок будь-якого рівня не повинне приводити до краху системи. Помилки повинні усуватися, діагностуватися і (якщо їх неможливо виправити) перетворюватися в коректні відмовлення. Системні структури даних повинні зберігатися безумовно. Збереження цілісності користувальницьких даних бажано.
4.Можливість супроводу. Можливі цілі супроводу - адаптація ПЗ до конкретних умов застосування, усунення помилок, модифікація. В усіх випадках потрібне ретельне структурування ПЗ і носієм інформації про структуру ПЗ повинна бути програмна документація про структуру ПЗ повинна бути програмна документація.
Адаптація в багатьох випадках може бути передовірена користувачеві - при ретельному відпрацьовуванні й описі сценаріїв інсталяції і настроювання.
Виправлення помилок вимагає розвинення сервісної служби, яка збирає інформацію про помилки і формуючі пакети, що виправляють.
Модифікація припускає зміну специфікацій на ПЗ. При цьому, як правило, повинні підтримуватися і старі специфікації. Еволюційний розвиток ПЗ заощаджує вкладення користувачів.

Системне програмування

Визначення (ДСТ)
Системна програма - програма, призначена для підтримки працездатності СОІ або підвищення ефективності її використання.

Визначення (ДСТ)
Прикладна програма - програма, призначена для рішення задачі або класу задач у визначеній області застосування СОІ.

Відповідно до термінології, системне програмування - це процес розробки системних програм (у т.ч., які керують і обслуговують).
З іншого боку, по визначенню Гегеля система - єдине ціле, що складається з безлічі компонентів і безлічі зв'язків між ними. Тоді системне програмування - це розробка програм складної структури.
Ці два визначення не суперечать один одному, тому що розробка програм складної структури ведеться саме для забезпечення працездатності або підвищення ефективності СОІ.
Зафіксоване в ДСТ підрозділ ПЗ на системне і прикладне є до деякої міри застарілим.
Сьогоднішній розподіл передбачає щонайменше три градації ПЗ:

- Системне
- Проміжне
- Прикладне

Проміжне ПО (middleware) ми визначаємо як сукупність програм, що здійснюють керування вторинними (розробленим самим ПЗ) ресурсами, орієнтованими на рішення визначеного (широкого) класу задач. До такому ПЗ відносяться менеджери транзакцій, сервери БД, сервери комунікацій і інші програмні сервери. З погляду інструментальних засобів розробки проміжне ПЗ ближче до прикладного, тому що не працює на пряму з первинними ресурсами, а використовує для цього обслуговування, надані системним ПЗ. З погляду алгоритмів і технологій розробки проміжне ПЗ ближче до системного, тому що завжди є складним програмним виробом багаторазового і багатоцільового використання й у ньому застосовуються ті ж або подібні алгоритми, що й у системному ПЗ.
Сучасні тенденції розвитку ПЗ складаються в зниженні обсягу як системного, так і прикладного програмування. Основна частина роботи програмістів виконується в проміжному ПЗ. Зниження обсягу системного програмування визначено сучасними концепціями ОС, об’єктної–орієнтованої архітектурою й архітектурою мікроядра, відповідно до яких велика частина функцій системи виноситься в утиліти, які можна віднести і до проміжного ПЗ. Зниження обсягу прикладного програмування обумовлене тим, що сучасні продукти проміжного ПЗ пропонують усе більший набір інструментальних засобів і шаблонів для рішення задач свого класу.

Значна частина системного і практично все прикладне ПЗ пишеться на мовах високого рівня, що забезпечує скорочення витрат на їхню розробку, модифікацію і переносність.

Системне ПЗ підрозділяється на системні керуючі програми і системні обслуговуючі програми.

Визначення (ДСТ)
Керуюча програма - системна програма, що реалізує набір функцій керування, яка містить у собі керування ресурсами і взаємодію з зовнішнім середовищем СОІ, відновлення роботи системи після прояву несправностей у технічних засобах.

Визначення (ДСТ)
Програма обслуговування (утиліта) - програма, призначена для надання послуг загального характеру користувачам і обслуговуючому персоналові СОІ.

Керуюча програма разом з набором необхідних для експлуатації системи утиліт складають операційну систему (ОС).

Крім вхідних до складу ОС утиліт можуть і входити й інші утиліти (того ж або іншого виробника), які виконують додаткове (опціонне) обслуговування. Як правило, це утиліти, що забезпечують розробку програмного забезпечення для операційної системи.

Визначення (ДСТ)
Система програмування - система, утворена мовою програмування, компілятором або інтерпретатором програм, представлених на цій мові, що відповідає документацією, а також допоміжними засобами для підготовки програм до форми, придатної для виконання.

Етапи підготовки програми

При розробці програм, а тим більше - складних, використовується принцип модульності, розбивки складної програми на складові частини, кожна з яких може розроблятися окремо. Модульність є основним інструментом структурування програмного виробу, що полегшує його розробку, налагодження і супровід.

Визначення (ДСТ)
Програмний модуль - програма або функціонально завершений фрагмент програми, призначений для збереження, трансляції, об'єднання з іншими програмними модулями і завантаження в оперативну пам'ять.

При виборі модульної структури повинні враховуватися наступні основні правила:

:

- Функціональність - модуль повинний виконувати закінчену функцію
- Незв'язність - модуль повинний мати мінімум зв'язків з іншими модулями, зв'язок через глобальні змінні й області пам'яті небажана
- Специфікуємість - вхідні і вихідні параметри модуля повинні чітко формулюватися

Програма пишеться у виді вихідного модуля, на малюнку - файл ІМ.

Визначення (ДСТ)
Вихідний модуль - програмний модуль вихідною мовою, оброблюваний транслятором і, що представляється для його як ціле, достатнє для проведення трансляції.

Першим (не для всіх мов програмування обов'язковим) етапом підготовки програми є обробка її Макропроцесором (або Препроцесором). Макропроцесор обробляє текст програми і на виході його отримуємо нову редакцію тексту. У більшості систем програмування Макропроцесор сполучений з транслятором, і для програміста його робота "не видна". Варто мати на увазі, що Макропроцесор виконує обробку тексту, це означає, з одного боку, що він "не розуміє" операторів мови програмування і "не знає" переміних програми, з іншої, всі оператори і змінні Макромови (тих виражень у програмі, що адресовані Макропроцесорові) у проміжному ЇМ' уже відсутні і для подальших етапів обробки "не видні". Так, якщо Макропроцесор замінив у програмі деякий текст А на текст В, те транслятор уже

бачить тільки текст В, і не знає, був цей текст написаний програмістом "своєю рукою" або підставлений Макропроцесором.

Наступним етапом є трансляція.

Визначення (ДСТ)
Трансляція - перетворення програми, представленої на одній мові програмування, у програму на іншій мові програмування, у визначеному змісті рівносильну першої.

Як правило, вихідною мовою транслятора є машинна мова цільової обчислювальної системи. (Цільова ОТ - та ОТ, на якій програма буде виконуватися.)

Визначення (ДСТ)
Машинна мова - мова програмування, призначена для представлення програми у формі, що дозволяє виконувати її безпосередньо технічними засобами обробки інформації.

Транслятори - загальна назва для програм, що здійснюють трансляцію. Вони підрозділяються на Ассемблери і Компілятори - у залежності від вихідної мови програми, які вони обробляють. ассемблери працюють з автокодами або мовами ассемблера, компілятори - з мовами високого рівня.

Визначення (ДСТ)
Автокод - символьна мова програмування, вирази якого по своїй структурі в основному подібні командам і оброблюваної даним конкретної машинної мови.

Визначення (ДСТ)
Мова Ассемблера - мова програмування, що являє собою символьну форму машинної мови зможливостями, характерними для мови високого рівня (звичайно містить у собі макрозасобу).

Визначення (ДСТ)
Мова високого рівня - мова програмування, вирази і структура якої зручіі для сприйняття людиною.

Визначення (ДСТ)
Об'єктний модуль - програмний модуль, отриманий в результаті трансляції вихідного модуля.

Оскільки результатом трансляції є модуль мовою, близькому до машинного, в ньому вже не залишається ознак того, на якій вихідній мові був написаний програмний модуль. Це створює принципову можливість створювати програми з модулів, написаних на різних мовах. Специфіка вихідної мови може позначатися на фізичному представленні базових типів даних, способах звертання до процедур/функцій і т.д. Для сумісності різномовних модулів повинні дотримуватися загальні положення.

Велика частина об'єктного модуля - команди і дані машинної мови саме в тій формі, у якій вони будуть існувати під час виконання програми. Однак, програма в загальному випадку складається з багатьох модулів. Оскільки транслятор обробляє тільки один конкретний модуль, він не може належним образом обробляти ті частини модуля, у яких запрограмовані звертання до даних або процедур, визначених в іншому модулі. Такі звертання називаються зовнішніми посиланнями. Ті місця в об'єктному модулі, де утримуються зовнішні посилання, транслюються в деяку проміжну форму, що підлягає подальшій обробці. Говорять, що об'єктний модуль являє собою програму машинної мови з недозволеними зовнішніми посиланнями.

Дозвіл зовнішніх посилань виконується на наступному етапі підготовки, що забезпечується Редактором Зв'язків (Компонувачем). Редактор Зв'язків з'єднує всі об'єктні модулі, що входять у програму. Оскільки Редактор Зв'язків "бачить" всі компоненти програми, він має можливість

обробити ті місця в об'єктних модулях, які містять зовнішні посилання. Результатом роботи Редактори Зв'язків є завантажувальний модуль.

Визначення (ДСТ)
Завантажувальний модуль - програмний модуль, представлений у формі, придатної для завантаження в оперативну пам'ять для виконання.

Завантажувальний модуль зберігається у виді файлу на зовнішній пам'яті. Для виконання програма повинна бути перенесена (завантажена) в оперативну пам'ять. Іноді при цьому потрібно деяка додаткова обробка (наприклад, настроювання адрес у програмі на ту область оперативної пам'яті, у яку програма завантажилася). Ця функція виконується Завантажником, що звичайно входить до складу операційної системи.
Можливий також варіант, у якому редагування зв'язків виконується при кожному запуску програми на виконання і сполучається з завантаженням. Це робитьм Звязувальний Завантажник. Варіант зв'язування при запуску більш витратний, тому що витрати на зв'язування тиражуються при кожному запуску. Але він забезпечує:

- велику гнучкість у супроводі, так як дозволяє змінювати окремі об'єктні модулі програми, не змінюючи інших модулів;
- економію зовнішньої пам'яті, так як об'єктні модулі, які використовувалися в багатьох програмах не копіюються в кожний завантажувальний модуль, а зберігаються в одному екземплярі.

Варіант інтерпретації пов’язаний з прямим виконанням вихідного модуля.

Визначення (ДСТ)
Інтерпретація - реалізація змісту деякого синтаксично закінченого тексту, який представлений конкретною мовою.

Інтерпретатор читає з вихідного модуля чергову пропозицію програми, переводить його в машинну мову і виконує. Всі витрати на підготовку тиражуються при кожному виконанні, отже, програма, що інтерпретується, принципово менш ефективна, ніж трансльована. Однак, інтерпретація забезпечує зручність розробки, гнучкість у супроводі і переносимість.
Приклади інтерпретаторів: мови процедур (sell, REXX), JVM.
Не обов'язково підготовка програми повинна вестися на тій же обчислювальній системі і в тій же операційній системі, в якій програма буде виконуватися. Системи, що забезпечують підготовку програм у середовищі, відмінної від цільової називаються кросс-системами. У крос-системі може виконуватися вся підготовка чи окремі її етапи:

- Макрообробка і трансляція
- Редагування зв'язків
- Налагодження

Типове застосування кросс-систем - для тих випадків, коли цільове обчислювальне середовище просто не має ресурсів, необхідних для підготовки програм, наприклад, вбудовані системи.
Програмні засоби, що забезпечують налагодження програми на цільовій системі можна також розглядати як окремий випадок кросс-системи.
Подальша тематика курсу відповідає планові наміченому в першій темі: ми будемо послідовно розглядати системні обробні програми, які забезпечують підготовку програм.

АЛГОРИТМІЧНЕ ПРОГРАМУВАННЯ

Історія розвитку ОТ пов’язана з історією розвитку системного програмного забезпечення.
Сучасні комп'ютерні системи поряд із прикладним ПЗ завжди містять системне, яке забезпечує організацію обчислювального процесу. Історія системного програмного забезпечення пов'язана з появою першої развитої у сучасному розумінні ОС UNIX.
1965 - Bell labs розробляє операційну систему Multix - прообраз UNIX, що має далеко не всі складові сучасної системи. До того часу не існувало мобільних ОС (придатних на різні типи машин) і Multix також не була мобільною ОС.
1971 - написаний UNIX для роботи на наймогутнішій платформі того часу PDP - 11
1977 - Стає переносною системою, тому що переписана мовою С (AT&T system V)

1981 - платформа Intel починає різко нарощувати свої можливості. Колосальним проривом було створення 8088, потім 8086, 80286, etc. З'являється однокористувальницька ОС MS-DOS, на 10 років стала стандартом де-факто для користувачів персональних комп'ютерів. Але для машин із процесором Intel з'являються і версії UNIX.
Кінець 80 – початок 90 р. – апаратні засоби різко збільшують свою потужність. Microsoft створює нову ОС Windows NT і з'являється стандарт Win32. Пізніше з'являється Windows 95 – ОС для одного робочого місця, яка має багато можливостей NT, покликана витиснути MS-DOS. Апаратні засоби дозволяють створювати 64-бітні версії такої ОС, як UNIX, і незабаром вона з'являється і використовується на платформі Alpha фірми DEC. З нарощуванням потужності апаратних засобів системне програмне забезпечення стає усе більш витонченим з великими можливостями.
2. Загальна класифікація обчислювальних машин. Сучасні архітектурні лінії ЕОМ. Системне ПЗ і його місце в сучасній інформатиці.
ЕОМ є перетворювачами інформації. У них вихідні дані задачі перетворяться в результат її рішення. Відповідно до використовуваної форми представлення інформації машини поділяються на два класи: безперервної дії - аналогові і дискретні дії - цифрові. У силу універсальності цифрової форми представлення інформації цифрові електронні обчислювальні машини являють собою найбільш універсальний тип пристрою обробки інформації. Основні властивості ЕОМ - автоматизація обчислювального процесу на основі програмного керування, величезна швидкість виконання арифметичних і логічних операцій, можливість зберігання великої кількості різних даних, можливість рішення широкого кола математичних задач і задач обробки даних. Особливе значення ЕОМ полягає в тому, що вперше з їхньою появою людина одержала знаряддя для автоматизації процесів обробки інформації. Керуючі ЕОМ – призначені для керування об'єктом або виробничим процесом. Для зв'язку з об'єктом їх постачають датчиками. Безперервні значення сигналів з датчиків перетворяться за допомогою аналогово-цифрових перетворювачів у цифрові сигнали, які вводяться в ЕОМ у відповідністю алгоритмом спрощення. Після аналізу сигналів формуються керовані впливи, які за допомогою цифро-аналогових перетворювачів перетворяться в аналогові сигнали. Через виконавчі механізми змінюється стан об'єкта.
Універсальні ЕОМ – призначені для рішення великого кола задач, склад яких при разаработке ЕОМ не конкретизується.

Приклад сучасних архітектурних ліній ЕОМ: персональні ЕОМ (IBM PC і Apple Macintosh – сумісні машини), машини для обробки специфічної інформації (графічні станції Targa, Silicon Graphics), великі ЕОМ (мэйнфрейми IBM, Cray, ЄС ЕОМ).

Загальне призначення системного ПО - забезпечувати інтерфейс між програмістом або користувачем і апаратною частиною ЕОМ (операційна система, програми-оболонки) і виконувати допоміжні функції (програми-утиліти) Сучасна операційна система забезпечує наступне:
1) Керування процесором шляхом передачі керування програмам.
2) Обробка переривань, синхронізація доступу до ресурсів.
3) Керування пам'яттю.
4) Керування пристроями введення-висновку.
5) Керування ініціалізацією програм, міжпрограмні зв'язки.
1) Керування даними на довгострокових носіях шляхом підтримки файлової системи.
Див. також стандарти в (1).
1. Загальне поняття архітектури. Принципи побудови ОТ 4-го покоління.
Архітектура – сукупність технічних засобів і їх конфігурацій, за допомогою яких реалізована ЕОМ. ЕОМ 4 покоління, має, як правило, шинну архітектуру, що означає підключення всіх пристроїв до однієї електричної магістралі, наз. шиною. Якщо пристрій виставило сигнал на шину, інші можуть його зчитати. Ця властивість використовується для організації обміну даними. З цією метою шина розділена на 3 адреси – шина адреси, шина даних і шина керуючого сигналу. Усі сучасні ЕОМ також включають пристрій, наз. арбітром шини, що визначає черговість заняття ресурсів шини різними пристроями. У PC поширені шини ISA, EISA, PCI, VLB.
Процесор (ЦП) – пристрій, що виконує обчислювальні операції і керує роботу машини. Містить пристрій керування, який вибирає машинні команди з пам'яті і виконує їх, і арифметико-логічний пристрій, що виконує арифметичні і логічні операції. Робота всіх електронних будов машини координується сигналами, вироблюваними ЦП. У сучасних ПК процесор представлений однієї СБИС, що містить понад мільйон транзисторів.

Оперативна пам'ять – призначена для збереження програм і даних, якими вони маніпулюють. Фізично виконана у виді деякого числа мікросхем. Логічно ОП можна представити як лінійну сукупність осередків, кожна з яких має свій номер, названий адресою. Час запису і читання з ОП у сучасних машинах займає частки мікросекунди, а для інших пристроїв цей час у 10-1000 разів більше.
Зовнішні пристрої – пристрої введення і виведення інформації. Оскільки, як правило, вони працюють значно повільніше інших, керуючий пристрій повинний призупиняти програму для завершення операції введення-виведнння з відповідним пристроєм.
1. Програмна модель ЕОМ. Основний командний цикл процесора. Поняття системи команд.
2. Адресація. Дані в ЕОМ: структура і формати представлення.
Адресація на прикладі процесора 8086.
Числа, встановлені процесором на адресній шині, є адресами, тобто номерами осередків оперативної пам'яті, з яких необхідно зчитувати чергову команду або дані. Розмір осередку оперативної пам'яті складає 8 розрядів, тобто 1 байт. Оскільки процесор використовує 16-розрядні адресні регістри, то це забезпечує йому доступ до 65536 (FFFFh) байт або 64КО (1К=1024 байт) основної пам'яті. Такий блок безпосередньо адресуємої пам'яті називається сегментом. Будб-який адрес формується з адреси сегмента (завжди кратний 16) і адреси осередку всередині сегмента (цю адресу називають зсувом). На комп'ютерах, оснащених процесором 8086, оперативна пам'ять звичайно має розмір, рівний 64КО. Для того щоб працювати з пам'яттю такого розміру, процесор здійснює перерахування адрес за допомогою процедури, називаної обчисленням ефективної адреси (мал.2.3).

Фізична 20-розрядна адреса обчислюється додаванням зсунутого вліво на 4 розряди 16-розрядної адреси сегмента оперативної пам'яті зі значенням 16-розрядного зсуву відносно початку цього сегмента. Використовуючи 20-розрядні адреси, можна адресувати 1М оперативної пам'яті (1М=1024К=1048576 байт). У програмі на асемблері повна адреса записується у виді SSSS:0000, де SSSS значення сегмента; 0000 – значення зсуву. Ділянка оперативної пам'яті розміром 16 байт називається параграфом.

Дані в ЕОМ – підрозділяються на числові і нечислові.
Числові дані:
1) Цілі типи – для представлення цілих чисел.
2)Дійсні типи – для представлення раціональних чисел. Бувають:
а) з фіксованою крапкою;
б) з плаваючою крапкою
Нечислові дані:
1) Логічні дані – приймаюче значення істина або неправда.
2) Строкові дані.
3) Множини.
4) Довільні дані (текст, звук, графіка).
1. Організація введення-виведення, класифікація зовнішніх пристроїв.
Організація введення-виведення в сучасних ЕОМ здійснена з використанням переривань. Це зв'язано з тим, що УВВ працюють набагато повільніше, ніж процесор і оперативна пам'ять. Тому керуючий пристрій повинний припиняти виконання програми і чекати завершення операції введення-виведення з зовнішнім пристроєм. При виведені всі результати виконаної програми повинні бути виведені на ВУ, після чого процесор переходить до чекання сигналів від ВУ. При введенні, наприклад, із клавіатури одержання значень натиснутих клавіш здійснюється при надходженні переривання від клавіатури.
2. Системні особливості архітектури ЕОМ. Приклади еволюції сучасних ВК – IBM 370, PDP11/VAX, Intel 80X86, RISC.
Системні особливості архітектуру ЕОМ полягають у відмінностях апаратних засобів, на яких реалізована машина.
Єдина система електронних обчислювальних машин (ЄС ЕОМ, аналог IBM 370) являє собою сімейство програмно - сумісних машин третього покоління. Кожна машин з сімейства складається з :
- процесора;
- оперативної пам'яті;
- каналів пристроїв, що забезпечують операції обміну даними між пам'яттю і зовнішніми пристроями незалежно від процесора;

- набору зовнішніх пристроїв введення-виведення, що виконують обмін інформацією між зовнішніми носіями і каналами.
Для ЄС ЕОМ характерна наявність каналів - спеціалізованих процесорів, які дозволяють звільнити процесор від виконання операцій введення-виведення і тим самим підвищити швидкість обміну з зовнішніми пристроями. У машинах сімейства ЄС за допомогою каналів забезпечується рівнобіжна робота процесора і зовнішніх пристроїв, а також рівнобіжне виконання операцій введення-виведення з декількома зовнішніми пристроями.
В основу побудови ЄС ЕОМ покладений принцип модульності, що дозволяє за бажанням користувача нарощувати обчислювальну потужність (міняти процесори), розширювати ємність оперативної пам'яті, додавати зовнішні пристрої.
Машини мають великі набори команд, розвинуте системне програмне забезпечення, що включає транслятори мов програмування Асемблер, ФОРТРАН, ПЛ/1, КОБОЛ, АЛГОЛ, ПАСКАЛЬ, операційні системи з різними функціональними можливостями.
Основна особливість керуючих обчислювальних машин типу PDP-11 полягає в тім, що взаємодія між усіма пристроями, що входять до складу комплексів , включаючи процесор , і оперативний запам'ятовуючий пристрій (ОЗУ) здійснюється за допомогою єдиного уніфікованого інтерфейсу, що одержав назву "Загальна шина" (ЗШ). Загальна шина є каналом, через який передаються адреси, дані, керуючі сигнали на пристрої комплексу, включаючи процесор і пам'ять.

Фізично ЗШ являє собою високочастотну магістраль передачі даних, що складає з 56 ліній.
Процесор використовує встановлений набір сигналів для зв'язку з пам'яттю і для зв'язку з зовнішніми пристроями, завдяки чому в системі відсутні спеціальні команди введення-виведення.
Усі пристрої комплексу підключаються в ЗШ по єдиному принципі. Деяким регістрам процесора, регістрам зовнішніх пристроїв, що є джерелами або приймачами при передачі інформації, на ЗШ видаляється адреса. У програмах адреси регістрів пристроїв розглядаються як адреси комірок пам'яті, що дозволяє звертатися до них за допомогою адресних інструкцій. Так, програмування операцій висновку даних на зовнішній пристрій практично зводиться до пересилання цих даних по визначеній адресі.
VAX – 11 – більш розвинута машина, ніж PDP-11. Це 32-бітова машина з адресним простором понад 4М. Вона по архітектурі схожа на PDP-11, але має 2 шинних адаптери – адаптер загальної шини й адаптер масової шини. Усі сумісні з загальною шиною периферійні пристрої можуть бути підключені до неї, тоді як високошвидкісні пристрої можуть бути підключені до масової шини через власні контролери. VAX – скорочено від англійських слів “віртуальне адресне розширення”, тобто машина має віртуальну пам'ять і багатозадачність.
Звичайно персональні комп'ютери IBM PC складаються з трьох частин : - системного блоку;
- клавіатури;
- дисплея.
Системний блок містить всі основні вузли комп'ютера :
- електронні схеми, що керують роботою комп'ютера (мікропроцесор, оперативна пам'ять, контролери пристроїв і т.д.);
- блок живлення;
- нагромаджувачі для гнучких магнітних дисків;
- нагромаджувач на твердому магнітному диску.
До системного блоку можна підключити ряд додаткових пристроїв уведення - виведення. Крім клавіатури і монітора такими пристроями є:
- принтер - для висновку на друк текстової і графічної інформації;
- миша - пристрій, що полегшує введення інформації в комп'ютер;
- стример - для збереження даних на магнітній стрічці;
- модем - для обміну інформацією з іншими комп'ютерами через телефонну мережу;
- сканер - прилад для введення малюнків і текстів у комп'ютер.
3. Двійкове кодування інформації. Представлення елементарних типів даних: натуральні числа, цілі числа зі знаком, числа з плаваючою крапкою.

Стан “встановлений” і “скинутий” співпадає 2 цифрам двійкової системи числення, фундаментальної для ЕОМ. Ці цифри називаються бітами. Двійкове кодування – представлення даних послідовністю бітами. При двійковому кодуванні числової інформації ступінь двійки при кожній двійковій цифрі на одиницю більше, ніж у попередньої цифри. Приклад конструкції двійкового коду:
1012=(1*22)+(0*21)+(1*20)=510.
У такий спосіб здійснюється представлення цілих чисел без знака. Негативні числа зі знаком представляються в додатковому коді – у формі доповнення до двох. Щоб знайти двійкове представлення негативного числа, треба взяти його позитивну форму, про інвертувати її і додати до отриманого результату 1.
Числами з плаваючою крапкою, називаються числа виду x=M*Pq, де M – мантиса, P – порядок, q – підстава системи числення. Нормалізована форма таких чисел:
M – дробове, |M| < 1.
Q – фіксовано.
P – ціле число зі знаком.
В машині в двійкових кодах зберігається M і P. Порядок, як правило, вирівнюється, тобто приводиться до більшого по модулі порядкові, шляхом зсуву мантиси вправо з меншим порядком на кількість розрядів, рівне (Pmax-Pmin)log2Q.

4. Правила арифметичних і логічних операцій з елементарними типами даних.
5. Властивості точності обчислень при роботі з плаваючою крапкою. Прийоми програмування, забеспеч. необхідну точність обчислень.
При роботі з числами з плаваючою крапкою, варто мати на увазі, що результат не цілком відповідає арифметиці дійсних чисел. Діапазон представлення чисел: -1*Qp<x<1*Qp. Для цих чисел поняття абсолютної точності заміняється на поняття відносна точність – це мінімальне число x, при якому R*(1+x) буде мати інший код, ніж число R. Втрата точності зв'язана із ситуацією поглинання менших операндів великими, котра у свою чергу зв'язана з фіксованим розміром мантиси. Спосіб збільшення точності – збільшення розміру мантиси (числа одинарної, подвійної, підвищеної точності в мовах високого рівня). При програмуванні операцій з плаваючою крапкою багатьох проблем можна уникнути, якщо правильно підібрати необхідні програмістові типи даних у залежності від розмірності і точності передбачуваних оброблюваних дійсних чисел. Деякі ситуації приводять до помилок обробки таких чисел:
а) Поглинання менших операндів великими.
б) Ситуація втрати точності (lost precision). Даний ефект виявляється, коли віднімаються 2 майже однакових числа.
в) Ситуація переповнення. У результаті операції порядок перевищує максимально припустиме значення.
г) Ситуація зворотна переповненню (underflow) – якщо порядок стає менше мінімального значення.
д) Втрата значимості – якщо мантиса стала дорівнювати нулеві при ненульовому порядку.
6. Представлення нечислової інформації. Текстові дані, символьні коди ASCII, EBSDIC, UNICODE. Особливості кодування російського алфавіту.
Будь-який текст являє собою послідовність літер літер 0..9 A..Z a..z А..Я а..я. Набір літер досить широкий за рахунок наявності національних мов. Перші машини мали алфавіт тільки з цифр і латинських букв. Ці літери утворили набір символів ASCII, EBSDIC. У цих кодуваннях загальна кількість символів не перевищує 128 (7 біт).
При представленні в пам'яті текст має вигляд послідовності байтів.
Способи представлення тексту
1. Фіксована довжина - |_|_|_|_|_|_|_|.
2. ASCIIZ - |_|_|_|_|_|_|0|. Кінець такого рядка позначає зарезервований символ (друкуване). Одержала широке поширення завдяки мові C.
3. Variable Lenght - |x|_|_|_|_|. Рядок містить довжину і послідовність байтів цієї довжини. Поширено в мові PASCAL.
Для представлення національних мов, зокрема російської, 128 символів у загальному випадку недостатньо. Тут використовується:
1) Розширення ASCII – використання 8 біта, тепер можна кодувати 256 символів;
2) UNICODE – використання 7 бітів, але більш ніж 1 символу кодування для представлення більшої кількості літер (наприклад для сумісності з мережами, що відтинають 8-й біт при передачі).

В даний час майже повсюдне використовується 8-бітове кодування символів. Кодова таблиця – графічне представлення символів, по яких можна визначити код. Проблеми при представленні російського алфавіту - а) необхідність сортування по кодах; б) при цьому треба залишити на старих місцях символи малювання рамок і заповнення (псевдографіки) для сумісності з іноземними програмами. Російське кодування – основна ДСТ – мала розташування символів за алфавітом, але в ній були зміщені символи псевдографіки. В даний час використовується альтернативне кодування ДСТ – у ній псевдографіка залишена на старому місці, але малі букви російського алфавіту розірвані (160-175, 224-239, 240-241). Це небагато утрудняє сортування – єдиний недолік. Крім того, є й інші кириличні кодування – МІС, КОИ-8, ISO-8859, т.п.

1. Представлення графічної інформації – растрове і векторне представлення, дозволяюща здатність, напівтонові і кольорові зображення, палітри.

Використання ЕОМ в автоматизованих системах управління, різних інформаційно-обчислювальних системах, системах колективного користування (див. гл. 13) вимагає їхнього укомплектування зручними засобами зв'язку людини з машиною. Одним з таких засобів є пристрій введення-виведення з електронно-променевою трубкою (ЭЛТ), назване монітором. В залежності від типу монітора на екран може виводитися як алфавітно-цифрова, так і графічна інформація. Пристрій висновку графічної інформації складається з відео пам’яті (буфера образа), монітора і пристрою сполучення, що передає на монітор зміст відео пам’яті. У сучасних машинах першому і третє об'єднано у відео адаптері.

При векторному представленні графічної інформації електронний промінь на моніторі безупинно пробігає між заданими точками, породжуючи відрізок – вектор. Таке представлення найбільш зручне для зображень, що складаються з ліній і простих геометричних фігур. У цьому випадку векторне зображення легке масштабується і вимагає малий обсяг пам'яті для збереження.

Якщо зображення складається з багатьох точок різних відтінків (напівтонові зображення), то векторний спосіб буде занадто складний у реалізації, і використовується растровий спосіб представлення – розбивка зображення на дрібні “клітки” і виведення на екран сітки крапок – растра (bitmap). Дозволена здатність растра – величина, що показує, скільки точок може бути виведене на квадратну одиницю зображення (од. виміру – dpi (точок на дюйм)). Для відеосистеми одиницею вимірювання може бути розмір виведеної точки і загальна кількість пікселів, виведене на екран монітора (наприклад, 1024x768).

У випадку монохромного зображення для кодування 1 точки в буфері образа досить одного біта – світиться чи ні. У випадку напівтонового або кольорового зображення кількість біт на крапку буфера повинне бути таке, щоб представити всі можливі кольори або відтінки. Наприклад, 8 бітами можна представити 256 квітів або відтінків. У таких системах програміст має доступ до палітри – ресурсів відеоадаптера, яка дозволяє встановлювати кольори або відтінки для кожного коду кольору.

2. Представлення звукової інформації – загальне поняття про дискретизацію і квантування звукових сигналів, точність представлення звукових коливань.

Представлення звукової інформації в ЕОМ:

а) Перетворення в цифрову форму за допомогою аналого-цифрового перетворювача. При цьому звук перетворюється в ланцюжок імпульсів, склда. з 8 або 16 біт (фактично з вектор 8-бітових іл 16-бітових чисел)

б) Дискретизація цього сигналу з постійною частотою. Якщо наприклад голос людини дискретизируется з частотою 8Кгц, використовуючи 8 біт, протягом 10с, це займе 80К. По дискретизированим значеннях можна відновити сигнал із заданою точністю і направити його в цифро-аналоговий перетворювач. Підсиливши сигнал з виходу ЦАП, одержимо звук.

При малій частоті дискретизації частина даних губиться внаслідок т.зв. шуму квантування, і ми не можемо досить точно відтворити дані. Закон Шеннона для дискретизації: для досягнення повної відповідності відновленого сигналу вихідному треба дискретизувать останній з частотою, у 2 рази перевищуючої його максимальну частоту. CD-програвачі працюють з частотою 44Кгц, і на такій частоті звук не втрачає якості.

Рівномірна дискретизація – заміна сигналу послідовністю його миттєвих значень, взятих через рівні проміжки часу.

Квантування – розбивка функції сигналу через визначений крок на інтервали - рівні квантування і заміна сигналу значеннями, узятими з цих інтервалів.

Формати збереження оцифрованного звуку у файлах:

VOC – стандарт де-факто від Creative.

WAV – формат оцифрованного звуку від Microsoft.

Layer 1, 2, 3 audio MPEG – ефективно закодований (стиснутий приблизно в 10 разів) формат збереження оцифрованного звуку - зараз здобуває найбільше поширення.
3. Загальний опис мови ASM. Типи даних.

Лекція 1

Мова Сі: історична довідка, загальна характеристика, основні достоїнства . Підготовка до виконання і виконання програм в операційному середовищі MS DOS. Елементи мови Сі: множини символів, ключові слова, константи і змінні, операції й оператори, керуючі конструкції. Приклади найпростіших програм: розбір елементів мови.

1. Мова Сі: Загальна характеристика, історична довідка основні досягнення

Мова Сі є мовою програмування загального призначення. Поява мови тісна пов'язана з створенням операційної системи UNIX, тому що вона спеціально розроблялася для цієї системи і на ній написана основна частина системного програмного забезпечення UNIXа. Однак мова Сі не орієнтована на роботі лише в одній операційній системі чи на якій-небудь конкретній машині. Вона являє собою універсальну, машинонезалежну, легко переносною мовою програмування, в однаковій мірі підходящий як для системного програмування, так і для рішення задач обчислювальної математики, технічних і комерційних задач. Мова Сі - це алгоритмічна мова "не дуже високого рівня". Приведена характеристика означає, що вона дає можливість роботи з такими типами об'єктів і дозволяє виконувати такі операції, що традиційно відносять до машиноорієнтированих мов "низького рівня", подібно мові асемблера. Прикладом згаданих об'єктів можуть служити символи, числа (зі знаком і без), адреси оперативної пам'яті і покажчики, бітові ланцюжки і т.д., над якими можна виконувати арифметичні, логічні і побіттові операції. У той час, Сі не забезпечує можливості роботи з рядками, множинами, масивами або списками як з єдиним цілим. У мові Сі є великий набір керуючих конструкцій для реалізації циклічних і розгалужених алгоритмів, засоби для блокового і модульного програмування, а також можливість гнучкого керування процесом виконання програми. Усе це дозволяє охарактеризувати Сі, як мову структурного програмування. Однак сама по собі мова Сі не забезпечує можливості керування введенням/ виведенням і в ньому немає операторів, подібних до операторів READ і WRITE у FoRTRaNe. Все же дія по обміні інформацією з периферійними пристроями виконуються за допомогою викликуваних зовнішніх функцій, що входять до складу стандартних бібліотек. Появу Сі прийнято пов'язувати з ім'ям Мартіна Рітчі, розробивши в 1972 році першу версію цієї мови в ході робіт над операційною системою UNIX для ЕОМ сімейства PDP. Однак історичне його виникнення варто пов'язувати з машинозалежною, мовою B, створена Кеном Томпсоном на основі мови BCPL. В даний час з 13000 рядків системного коду UNіха лише 800 рядків, що виконують роботу найбільш низького рівня, написано на мові асемблера. Інша ж частина цієї операційної системи і безліч програмних утиліт написано на Сі. На закінчення перелічимо деякі основні властивості мови Сі:

- широкий набір керуючих конструкцій для організації циклів і умовних переходів, які забезпечують можливість написання гнучких і добре структурованих програм;
- великий набір операторів і операцій, багато хто з яких допускають пряму трансляцію в машинний код;
- різноманіття примітивних типів даних, включаючи можливість керування довжиною цілих і дійсних змінних;
- наявність засобів для конструювання нових агрегатированих типів даних;
- можливість безпосередньої роботи з машинними адресами через апарат покажчиків;
- наявність препроцесора, що дозволяє вносити зміни в текст програми безпосередньо перед її компіляцією;
- блокова структура програмного коду, можливість роботи з різними класами пам'яті і засобів для модульного програмування.

2. Підготовка до виконання і виконання програм

Мова Сі відноситься до числа компільованих мов програмування. Це означає, що підготовка до виконання Сі-програми містить у собі наступні етапи:

- введення вихідного тексту програми у файл за допомогою будь-якого редактора текстів (ім'я файлу, як правило, має стандартне розширення "C");
- компіляція програми, тобто перетворення її опису вхідною мовою в семантичний еквівалент машинною мовою, названий об'єктним модулем (ім'я файлу, в якому міститься результат компіляції, звичайно має розширення "OBJ");

побудова готового до виконання завантажувального модуля з об'єктних модулів, включаючи модулі з зовнішніх бібліотек (файл, який містить готову програму, має ім'я з розширенням "EXE"). У процесі компіляції програми створюється текст її вихідного тексту, що містить, можливо, повідомлення про виявлені помилки.

Наявність текст істотно спрощує пошук і усунення помилок, допущених при підготовці програми, і скорочує час, затрачуваний на її налагодження. Ім'я файлу, у якому записується текст вихідної програми, має звичайно розширенням "LST". На етапі побудови завантажувального модуля також можливе створення файла-текста, що включає в себе інформацію про розміщення об'єктних модулів, які розміщуються, в пам'яті ЕОМ. Однак його аналіз і використання вимагає більш глибоких знань основних принципів роботи машини, ніж розбір лістинга вихідної програми. Для запуску готової до виконання програми в операційному середовищі MS DOS досить набрати ім'я файлу на клавіатурі консольного термінала, закінчивши введення натисканням клавіші Enter.

3. Елементи мови Сі

У цьому параграфі в стиснутій формі приводяться основні відомості про структурні елементи мови Сі й обговорюються загальні питання організації програми.

3.1. Алфавіт

3.1.1. Букви і цифри

- великі букви латинського алфавіту

A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z

- малі букви латинського алфавіту

a b c d e f g h i j k l m
n o p q r s t u v w x y z

- десяткові цифри

0 1 2 3 4 5 6 7 8 9

3.1.2. Спеціальні символи

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

3.2. Імена об'єктів програми (ідентифікатори)

Ідентифікаторами в мові Сі є послідовності букв і цифр, які починаються з букви, причому символ підкреслення (_) розглядається компілятором як буква. Великі і малі букви латинського алфавіту вважаються різними. Довжина ідентифікатора формально може бути довільної, однак тільки лише перші 31 символ є значимими для компілятора IBM C/2. У програмах на мові Сі ідентифікатори служать

іменами змінних, символічних констант, функцій, типів даних і міток. Загальноприйняте для іменування констант використовувати великі букви латинського алфавіту, використовуючи малі букви для позначення змінних і функцій.
Зауваження. Не рекомендується в прикладних програмах використовувати ідентифікатори, що починаються з підкреслення, тому що в середовищі IBM C/2 це може привести до конфлікту з іменами створюваних компілятором службових змінних або констант.

3.3. Ключові слова

Ключові слова - це визначені ідентифікатори, що мають спеціальні значення. Імена ніяких об'єктів програми не повинні збігатися з наступними зарезервованими ключовими словами:
- типи даних

```
char    float   short   typedef   void
double  int     signed  union
enum    long    struct  unsigned
```

- класи пам'яті

```
auto   extern   register   static
```

- оператори

```
break    default   for    return   while
case     do        goto   sizeof
continue else      if     switch
```

- спеціальні

```
const   far     huge    pascal
cdecl   fortran near    volatile
```

3.4. Коментарі

Під коментарями розуміються послідовності символів, які ігноруються компілятором. Коментарі мають наступну форму:

```
/* Characters */
```

де Characters є довільна комбінація будь-яких друкованих символів ASCII.

3.5. Константи

Константами в мові Сі можуть бути числа (цілі і дійсні), символи і рядки символів, які припустимо використовуються в програмі в змісті їхніх значень. Значення константи не може бути змінене в процесі роботи програми. Спеціальні директиви мови дають можливість присвоювати константам будь-якого типу символічні імена.

3.6. Операції і вирази

Під операціями варто розуміти дії, виконувані програмою над визначеними в ній об'єктами. Для позначення операцій використовуються деякі стандартні комбінації спеціальних символів з алфавіту мови. Об'єкти програми, що беруть участь в операції, називаються операндами. Будь-яка комбінація одного або більшого числа оперантів і символів операцій, що дає єдине значення, утворюють вирази.

Будь-які вирази, що закінчується крапкою з коми, є оператором (див. п. 3.7.). Всі операції на мовиіСи зручно розділити на вісім наступних груп:
- арифметичні операції

+ - * / % ++ -i

- операції відношення

== != < > <= >=

- логічні операції

! || &&

- побітові операції

~ << >> & | ^

- операції присвоювання

= += -= *= /= %= <<= >>= &= ^= |=

- операції над масивами

[]

- операції над структурами й об'єднаннями

->

- інші операції

? : , syzeof (type specifier) ()

3.7. Оператори

Найпростішим оператором мови Сі є будь-який вираз, що закінчується крапкою з коми (див. п. 3.6.). Зокрема, що окремо стояча крапка з коми інтерпретується компілятором як нуль-оператор. Найбільш значну групу операторів утворять інструкції (оператори) керування процесом виконання програми. До них відносяться:
- оператори циклу

for (<init-expression>; <cond-expression>;
<loop-expression>)
statement
while (expression)
statement

do
statement
while (expression);

- умовний оператор і перемикач

```
if (expression)
  statement1
<else
  statement2>

switch (expression)
{ <case  const-expression: <statements>>
  .                        .
  .                        .
  .                        .
 <default: <statements>>
  .                        .
  .                        .
  .                        .
 <case  const-expression: <statements>> }
```

- оператори передачі керування

```
goto    break    continue    return
```

Тут expression є довільне правильне вираження мови Сі (див. Лекцію 2), а через statement позначений простий або складений (див. нижче) оператор, яким, зокрема, може бути і будь-який оператор керування. Останнє означає, що припустимим є вкладення операторів керування один в одного. Елементи конструкцій мови, закладені в кутові дужки (<>), не є обов'язковими і можуть бути опущені. Один чи декілька операторів, закладені у фігурні дужки, утворюють складений оператор або блок. Крапка з комою після закриваючої дужки при цьому не ставиться.

3.8. Опис

Опис в мові Си - це рядка програми, що визначають імена і характеристики функцій, змінних, типів і символічних констант, використовуваних у програмі.

3.9. Загальна структура програми

Всяка програма на мові Сі представляє собою сукупність функцій, які виконують основну роботу по реалізації деякого алгоритму. Кожна з цих функцій, у свою чергу, є незалежний набір описів і операторів, вкладених між заголовком функції і її кінцем. Та функція, з якої починається виконання програми, називається головною функцією. Вона повинна мати визначене ім'я main().

Лекція 2

Поняття типу даних. Змінні і константи. Основні типи даних у мові Сі: загальна характеристика, машинне представлення, опис даних у програмі. Числові, символьні і строкові константи. Арифметичні операції й арифметичні вирази. Операції відношення, логічні операції і логічні вирази. Умовна операція. Автоматичне перетворення типів і операція приведення. Найпростіші оператори мови Сі. Складений оператор.

1. Поняття про тип даних. Змінні і константи. Операція присвоювання

Всяка програма, призначена для реалізації на ЕОМ, представляє собою формалізований опис алгоритму рішення тієї або іншої задачі. Дії, виконувані програмою відповідно до цього алгоритму, спрямовані на перетворення деяких об'єктів, що визначають поточний стан процесу рішення задачі. Такі внутрішні об'єкти програми прийнято називати даними. Для збереження всякого елемента даних виконуюча

система виділяє необхідний простір в оперативній пам'яті ЕОМ, розмір якого ми будемо називати довжиною цього елемента. Найважливішою характеристикою будь-якого елемента даних є його тип. Поняття типу містить у собі наступну інформацію про елемент даних:
- припустимий набір значень, який об'єкт цього типу може приймати в процесі роботи програми (сукупність усіх зазначених значень ми будемо називати областю визначення типу);
- склад операцій, що дозволено виконувати над об'єктами даного типу;
- спосіб представлення елемента даних в пам'яті машини;
- правила виконання операції з припустимого для цього типу набору операцій.

Мова Сі забезпечує можливість представлення і обробки даних наступних основних типів:
- цілі числа різної довжини зі знаком і без;
- дійсні числа різної довжини;
- символи, представлені у форматі стандарту ASCII.

Ті елементи даних, що зберігають незмінні значення протягом всього часу роботи програми, прийнято називати константами. Інші ж об'єкти, які є предметом змінення в ході виконання алгоритму, називають змінними. З погляду мови Сі, всяка змінна величина ототожнюється з її ім'ям, або ідентифікатором. З позиції ж ЕОМ, вона розглядається як змінюється в часі вміст деякої області оперативної пам'яті. Інструкція мови програмування, що дозволяє призначати і змінювати значення змінних, називається операція присвоювання. Для її позначення в Сі використовується символ "=". Наприклад:

a = 2; b = 3*(a+4);

Після виконання зазначених дій змінна a приймає значення, рівне 2, а змінна b - значення 18.

2.Типи даних у мові Сі. опис даних у програмі

Як вже говорилося, опис в мові Сі - це рядки програми, що визначають імена і характеристики елементів даних, які беруть участь у роботі алгоритму. У найпростішому випадку інструкція опису даних у Сі-програмі має наступний формат:

<sc-specifier> type-specifier identifier <, identifier ...>;

де sc-specifier є опис класу пам'яті (див. Лекцію 7, $ 2);
type-specifier - опис типу;
identifier - ім'я (ідентифікатор) змінної.

Для більшості компіляторів мови С припустимими є використання типів, які показані в наступній таблиці. В таблиці в круглих дужках зазначена довжина елемента даних кожного типу в байтах (1 байт = 8 біт) і область припустимих значень для персонального комп'ютера IBM PC AT.

Тип	Семантика і довжина	Діапазон
Char	Символьна зі знаком (1)	від -128 до 127
Int	Ціла (2)	від -32768 до 32767
Short	Коротка ціла(2)	від -32768 до 32767
Long	Довга ціла (4)	від -2147483648 до 2147483647
unsigned char	Символьна без знака (1)	від 0 до 255
unsigned	Ціла без знака(2)	від 0 до 65535
unsigned short	Коротка ціла без знака (2)	від 0 до 65535
unsigned long	Довга ціла без знака (4)	від 0 до 4294967297
float	Дійсна (4)	|x| >= 1.0E-38 |x| <= 1.0E+38
double	ДІйсна подвійної точності (8)	|x| >= 1.0E-308 |x| <= 1.0E+308

Зауваження. Ключові слова unsigned, signed, short і long можуть бути використані як модифікатори основних типів даних при побудові похідних. Наприклад, наступні описи розглядаються компілятором як еквівалентні:

short int ..... short
long int ..... long
unsigned int ..... unsigned
unsigned short int ..... unsigned short
unsigned long int ..... unsigned long
signed char ..... char
signed int ..... int
signed short ..... short
signed long ..... long

Розглянемо деякі приклади опису даних у програмі.

```
int    a, b, c;       /*  Змінні a, b, c оголошені  */
                      /*  мають тип int             */
float  alpha, beta;   /*  Змінні alpha і beta оголошені  */
                      /*  мають тип float           */
```

Оскільки внутрішні машинні представлення даних тих самих типів для різних ЕОМ можуть бути різними, то виникають труднощі при необхідності забезпечити належну мобільність програмного забезпечення. Так, наприклад, всяка змінна типу int на машинах IBM PC AT займає два байти пам'яті, у той час як для її представлення на ЕОМ VAX-11 потрібно чотири байти.
Подібні розходження можуть привести до неправильної роботи програм, які маніпулюють, наприклад, бітовими ланцюжками або потребуючим динамічним виділенням пам'яті для збереження змінних. Перебороти виникаючі при цьому труднощі можна, використовуючи, де це необхідно, операцію визначення пам'яті, необхідної для представлення деякої змінної або якого-небудь типу. У загальному випадку ця операція виглядає в такий спосіб:

sizeof(name),

де name є або ідентифікатор змінної, або ім'я типу даних. У наступному прикладі

a = sizeof(int);

змінна a приймає значення, рівне кількості байт пам'яті, необхідних для представлення будь-якої величини типу int для конкретної ЕОМ.

3. Константи в мові С

Константами в мові Сі можуть бути числа (цілі і дійсні), символи і рядки символів, які дозволено використовувати в програмі в змісті їхніх значень. Нижче розглянуті припустимі формати запису констант чотирьох зазначених видів.

3.1. Цілі константи

Цілі константи можуть бути представлені в трьох основних форматах: десятковому, восьмеричному і шістнадцятирічному.
Десятковий формат:

<->dec_digits<L|l>,

де dec_digits є одна або більша кількість десяткових цифр (від 0 до 9).

Восьмеричний формат:

0oct_digits<L|l>

де oct_digits є одна або більш кількість восьмеричних цифр (від 0 до 7).
Шістнадцятирічний формат:

0xhex_digits<L|l> або 0Xhex_digits<L|l>

де hex_digits є одна або більш кількість шістнадцятирічних цифр (від 0 до F).

Зазвичай компілятор розглядає десяткові константи як числа зі знаком і присвоює їм тип int. Якщо значення константи перевищує найбільше машинне ціле зі знаком, то вона представляється як довге ціле (тип long). Знак мінус у записі десяткової константи інтерпретується компілятором як арифметичний оператор, а сама константа в цьому випадку є константним арифметичним виразом. Восьмеричні і шістнадцятирічні константи можуть бути типу int, unsigned, long або unsigned long у залежності від їхньої абсолютної величини. Якщо константа допускає представлення у виді цілого числа стандартної довжини, то компілятор присвоює їй тип int. Якщо ж значення константи перевищує максимальне позитивне число зі знаком, представлене як int, але менше числа, яке без обліку знака може бути представлено тим же числом розрядів, що і int, то їй присвоюється тип unsigned. Аналогічно, якщо константа не може бути представлена як unsigned, вона одержує тип long або unsigned long. Модифікатор L або l, що безпосередньо слідує за записом константи, вимагає від компілятора її представлення як має тип long або unsigned long незалежно від абсолютної величини. Наступна таблиця ілюструє різні форми запису цілих констант.

Десяткова	Восьмерична	Шістнадцятирічна
10	012	0xa або 0XA
132	0204	0x84
32179	076663	0x7DB3
15L	017L	0xFL

3.2. Константи з плаваючою крапкою

Константи з плаваючою крапкою являються дійсними десятковими числами зі знаком. Їхній запис може містити в собі цілу частину, десяткову крапку, дробову частину, символ експоненти e або E і показник експоненти у виді цілого числа зі знаком:

<->digits.<digits><E|e<+|->digits>

або

<-><digits>.digits<E|e<+|->digits>

де digits є одна або більша кількість десяткових цифр (від 0 до 9). Знак мінус перед константою розглядається компілятором як арифметичний оператор і тому дійсна негативна константа власне кажучи є константним арифметичним виразом. У записі дійсної константи може бути відсутня або ціла, або дробова частина (але не обидві відразу). Відсутність десяткової крапки припустимо лише при наявності експоненти. Наступні приклади ілюструють можливі форми запису дійсних констант:

15.75 1.575E1 1575e-2 .1575E2 -.175e2 0.137 137. 137.0

Будь-яка константа з плаваючою крапкою, розглядається компілятором як тип double, тобто представляється з подвійною точністю.

3.3. Символьні константи

Символьна константа являє собою букву, цифру, знак пунктуації або Esc-послідовність, закладені в одиночних лапках:

'character' або 'Escape sequence',

де character є будь-який друкований символ ASCII, крім одиночних лапок (') або зворотною косою рискою (\). Esc-послідовності складаються зі зворотної косої риски, за якої слідують буква або комбінація цифр. Символи ' і \ також представляються у виді Esc-послідовностей. У наступній таблиці приведені припустимі в мові Сі Esc-послідовності і їхні значення.

\a - звуковий сигнал \t - горизонтальна табуляція
\b - крок назад \v - вертикальна табуляція
\f - нова сторінка \' - одиночні лапки
\n - новий рядок \" - подвійні лапки
\r - повернення каретки \\ - зворотна коса риска
\ddd - символ ASCII у восьмеричному представленні
\xddd - символ ASCII у шістнадцятирічному представленні
В усіх інших випадках символ \ ігнорується компілятором

Послідовності виду \ddd і \xddd дозволяють представити будь-який символ ASCII, включаючи і символи, що недрукуються, у восьмеричному і шістнадцятирічном форматі відповідно. Незначні нулі ліворуч в обох випадках можуть бути опущені. Значенням символьної константи є ціле число, що відповідає внутрішньому машинному представленню відповідного символу. Нижче приведені приклади правильного запису символьних констант:

'A' 'c' '?' '7' '\"' '\\' '\n' 'r' '\033' '\10' '\x1F'

3.4. Рядки символів

Рядками в мові Сі є послідовності символів ASCII, в подвійних лапках:

"<characters>",

де characters є один або більша кількість символів ASCII, крім подвійних лапок (") і зворотню косу риску (\). Недруковані символи ASCII, а також символи " і \ у складі рядків повинні представлятися відповідними Esc-послідовностями. Компілятор IBM C/2 заносить символи рядка в пам'ять ЕОМ у порядку їхнього проходження, автоматично додаючи нуль-символ (Esc-послідовність \0) у кінець рядка. Формально припустимі є також порожній рядок (""), що складається у внутрішнім представленні з одного символу \0. Приведемо кілька прикладів правильного запису символьних рядків:

"Це рядок символів"
"Enter a number between 1 and 100 \n Or press Enter"
"\"Yes, I do\", she said."

Для формування довгої послідовності символів, які займають більше одного рядка на екрані термінала, необхідно набрати символ \, натиснути клавішу Enter і продовжити введення символів з нового рядка:

"Long string can be broken\
into two pieces."

Важливо замітити, що символьна константа, наприклад 'x', не ідентична рядкові "x", яка містить один символ. Справа в тім, що символьна константа займає один байт пам'яті і рівносильна своєму числовому кодові у внутрішнім машинному представленні. Рядок же з одного символу займає два байти пам'яті, перший з яких містить код цього символу, а другий - символ \0.

4. Арифметичні операції й арифметичні вирази

У Лекції 1 нами вже була розглянута одна з найважливіших операцій мови Сі - операція присвоювання, яка дозволяє призначати і змінювати значення змінних. Тут ми розглянемо групу арифметичних операцій і визначимо поняття арифметичного виразу. Традиційним способом задавання арифметичних операцій є використання двохмістних арифметичних операторів мультиплікативної й адитивної груп. Операндами всякої двохмістної операції можуть бути як константи, так і змінні, причому імена останніх повинні бути попередньо визначені в одній з інструкцій опису типу. Оператори мультиплікативної групи служать для представлення операцій множення (*), ділення (/) операндів і одержання залишку (%) від розподілу першого операнда на другий. В останньому випадку обидва операнди повинні бути цілими величинами. Операція множення здійснюється над операндами будь-яких (можливо різних!) типів. Операція ділення, застосована до двох цілих операндів, може привести до втрати дробової частини результату. Спроба ділення на нуль дає помилку на етапі компіляції або виконання програми. Порядок виконання операцій мультиплікативної групи – з ліва на право. Група адитивних операторів містить у собі два оператори: додавання (+) і віднімання (-). Обидві ці операції здійсненні над операндами будь-яких (можливо різних!) типів. Порядок виконання операцій адитивної групи - зліва направо. Їхній пріоритет нижче пріоритету мультиплікативних операцій, але вище, ніж пріоритет операції присвоювання.

Зауваження. Двумістну операцію віднімання варто відрізняти від операції що має найвищий пріоритет одномісної (унарної) операції зміни знака, символом який служить знак мінус (-) ліворуч від операнда.

Одномісними операціями яки набули найбільшого використання у програмах на мові С є операції збільшення (++) і зменшення (--) значення змінної на одиницю. Кожна з них має дві форми: префіксну і постфіксну. У префіксній формі символ операції ставиться ліворуч від свого операнда, а в постфіксній - праворуч від нього. Семантична відмінність двох цих форм полягає в тому, що у випадку префіксного запису (++x) збільшення або (--x) зменшення значення змінної x виконується до його використання в більш складному виразі, у той час як постфіксний запис (x++) або (x--) змінює колишнє значення змінної лише після його фактичного використання. Пріоритет операцій ++ і -- вище пріоритету будь-якої двохмістної арифметичній операції й операції присвоювання, а порядок їхньої обробки компілятором – з право наліво. Сукупність одного або більшої кількості чисел операндів і символів операцій (включаючи операцію присвоювання) утворить арифметичний вираз. При обчисленні значення виразу спочатку виконуються унарні операції (-, ++, --), потім двохмістні операції мультиплікативної (*, /, %) і аддитивних (+, -) груп і, нарешті, операції присвоювання. Звичайний порядок виконання операцій може бути змінений шляхом веденням деякої частини арифметичного виразу в круглі дужки. Приведемо кілька найпростіших прикладів арифметичних виразів.

1. a = 4.3 + 2.7

У цьому прикладі обидва операнди праворуч від оператора присвоювання є числовими константами. Арифметичні вирази такого виду прийнято називати константними вираженнями.

2. b = (c + d) % 4

Тут традиційний порядок виконання арифметичних операцій порушений використанням круглих дужок, які виділяють підвираз в складі більш складного арифметичного виразу. Відповідно до визначення операції % операнди c і d повинні бути змінними цілого типу.

3. e = ++f/(g + h)

В даному прикладі використана префіксна форма оператора ++ і тому операція ділення буде виконана після фактичного збільшення значення змінної f на одиницю.

Домашнє завдання: Приклади арифметичних виразів

Визначити значення всіх змінних після виконання наведених виразів чи вказати на неможливість використання даних виразів за умови наступних вихідних значень: a=3; b=10; c=21;

1) k1=-a*3%c;
2) k2=--b*a;
3) k3=b+++4;
4) k4=--c+++b;

Зауваження. У мові Сі операція присвоювання є повноправною частиною будь-якого арифметичного виразу й у загальному випадку має формат:

expression1 = expression2

Однак, не вдаючись поки в семантичні подробиці такого запису, ми будемо користуватися її спрощеною формою

identifier = expression

де identifier є попередньо визначене ім'я змінної, а expression - довільний арифметичний вираз. Але навіть у таку спрощену схему укладаються досить складні в семантичному відношенні вирази виду

p = 2*(q + s)/(t = u * ++v)

5. Операції відношення, логічні операції і логічні вирази

Група двохмістних операцій відношення виконує порівняння першого операнда з другим, перевіряючи, чи є істиною те співвідношення між операндами, що визначене символом операції. Повний набір операцій відношення задається наступною групою операторів:

< - менше > - більше
<= - менше або дорівнює >= - більше або дорівнює
== - дорівнює != - не дорівнює

Результатом будь-якої операції відношення є числове значення типу int, яке дорівнює одиниці, якщо порівняння істина, і нулеві в противному випадку. Таким чином, операції відношення у внутрішнім машинному представленні приводять до арифметичного результату.

Зауваження. Строго кажучи, логічне значення "істина" відповідає будь-якому числовому значенню, відмінному від нуля. Надалі ми побачимо, що саме така домовленість прийнята в мові Сі. Це дає можливість об'єднати поняття арифметичного, умовного і логічного виразів у єдиному понятті "вираз", що дуже важливо з точки зору гнучкості і "симетричності" мови.

Вирази, які сконструйовані за допомогою операцій відношення, прийнято називати умовними виразами. У процесі обчислення значення будь-якого умовного виразу, операції відносини обробляються з ліва на право. Встановлений порядок може бути змінений шляхом ведення частини виразу в круглі дужки. Наприклад, запис

x < y == z

цілком еквівалентна запису

(x < y) == z

у той час як вираз виду

x < (y == z)

відрізняється від попереднього порядком виконанням операцій < і ==. При розробці реальних програм часто виявляється необхідним об'єднати два або більш умовних виразів. Це можна зробити, використовуючи набір двохмістних логічних операцій:

&& - логічне І
|| - логічне АБО
! - логічне НІ (заперечення)

Припустимо, що expression1 і expression2 - два простих умовних вирази. Тоді:

1. значення expression1 && expression2 є істиною тоді і тільки тоді, коли обидва вирази expression1 і expression2 істинні;
2. значення expression1 || expression2 є істиною, якщо хоча б один з виразів-операндів має значення "істина";
3. значення !expression1 є істиною, якщо вираз expression1 є не істиною, і навпаки.

Вирази, побудовані з використанням логічних операцій, ми будемо називати логічними виразами. Помітимо, що логічні вирази є прямим узагальненням простих умовних виразів. Стандартний порядок їхньої обробки – з ліва на право. Пріоритет логічних операцій && і || нижче пріоритету будь-якої операції відношення і тому логічні вирази

a < b && b < c і (a < b) && (b < c)

цілком рівносильні, хоча друге з них є більш кращим через властивої йому наочності. Однак операція логічного заперечення (!) має дуже високий пріоритет (він такий же, як пріоритет одномісних арифметичних операцій) і тільки круглі дужки мають більш високий пріоритет.

Зауваження. У загальному випадку операндами логічних операцій можуть бути не тільки умовні вирази, але і будь-які арифметичні вирази. Це легко зрозуміти, якщо нульовому значенню арифметичного виразу поставити у відповідність логічне значення "не істина" і, навпаки, всяке відмінне від нуля числове значення ототожнити з логічним значенням "істина". До докладного розгляду даного питання ми повернемося в наступних розділах. Найпростішою інструкцією мови Сі, що використовує логічні вирази, є умовний оператор:

expression1 ? expression2 : expression3

де expression1 це логічне вираз, а expression2 і expression3 це довільні арифметичні вирази. Якщо expression1 приймає значення "істина", то результатом умовної операції буде значення expression2, у противному випадку він дорівнює значенню expression3. Наприклад, інструкція

abs_a = (a > 0) ? a : -a

привласнює змінній abs_a абсолютне значення змінної a.

6. Автоматичне перетворювання типів і операція приведення

Якщо до складу арифметичного або умовного виразу входять операнди різних типів, то компілятор автоматично виконує їхнє приведення до загального типу. Незважаючи на те, що в ряді випадків характер перетворення залежить від виду конкретної операції і типу операндів, існує загальний набір стандартних правил перетворення:

1. якщо операція виконується над даними двох різних типів, обидві величини приводяться до "вищого" типу;
2. в операторі присвоювання кінцевий результат обчислення виразу в правій частині приводиться до типу змінної, якій повинне бути привласнене значення.

Послідовність імен типів, упорядкованих від "вищого" типу до "нижчого", виглядає так: double, float, long, int, short і char. Застосування ключового слова unsigned підвищує ранг відповідного типу даних зі знаком.

Крім цього стандартний компілятор у всіх випадках розширює короткі операнди до операндів стандартної довжини, прийнятої при виконанні машинних операцій. Так, операнди, що мають тип float, перетворяться до типу double, операнди типу char і short - до типу int, а операнди типу unsigned char і unsigned short - до unsigned.
Поряд з автоматичним перетворенням типів, виконуваним виконуючою системою, у мові Сі мається можливість точно вказати тип даних, до якого необхідно привести деяку величину. Ця можливість реалізується в операції приведення типів у такий спосіб: перед даною величиною в круглих дужках записується ім'я необхідного типу. Нехай, наприклад, змінна res має тип int. Тоді значення арифметичного вираження
res = 2.7 + 1.5

відповідно до загальних правил перетворення типів, дорівнює 4. При застосуванні явної операції приведення типу до обох операндів у правій частині

res = (int)2.7 + (int)1.5

одержимо результат, рівний 3. Цей приклад говорить про необхідність з особливою обережністю відноситися до всіляких перетворень типів операндів.

7. Найпростіші оператори мови Сі. Складений оператор

Розглянуті вище арифметичні, умовні і логічні вирази в сукупності утворюють об’єкти програми, які прийнято називати просто виразами. Цією термінологією підкреслюється рівноправність операцій різних груп з погляду їхньої ролі у формуванні виразів. Більш того, усякому виразу в Сі відповідає деяке числове значення (за винятком, звичайно, тих випадків, коли результат операції вважається невизначеним), які може бути використане надалі. Однак самі по собі вирази будь-якого виду ще не є закінченими інструкціями мови Сі. Для позначення завершених дій використовується поняття оператора. Найпростішим оператором мови є будь-який вираз, що закінчується крапкою з коми (;). Зокрема, окремо поставлена крапка з комою інтерпретується компілятором як нуль-оператор. Наприклад, інструкції виду

```
a = c*(d + e)/4;
f = alpha++;
```

являються виразами-операторами, відіграють роль закінчених розпоряджень для виконуючої системи. Група з одного або більшого числа операторів, закладених у фігурні дужки, утворюють складений оператор. Помітимо, що за правилами мови крапка з комою після закриваючої фігурної дужки не ставиться. Наприклад, група з трьох операторів

```
{ x = 7; y = x + 10; z = (x + y)/13; }
```

розглядається компілятором як один складений оператор. У програмі мовою Сі складені оператори можуть бути використані усюди нарівні з простими операторами.
Зауваження. У літературі складений оператор часто називають блоком. Ми ж тут будемо розрізняти ці поняття, припускаючи повернутися до другого їх у Лекції 7.

Лекція 3

Масиви змінні як однорідні статичні структури даних. Рядки символів. Ініціалізація змінних і масивів.Керуючі конструкції мови Сі: синтаксис і семантика.

$ 1. Масиви змінних як однорідні статичні структури даних

На початку попередньої лекції ми познайомилися з основними типами даних мови Сі і правилами опису змінних у програмах. Однак дотепер нам приходилося мати справа лише з простими змінними, які є

носіями єдиного значення, чи число або внутрішнє машинне представлення деякого символу ASCII. Однак при рішенні на ЕОМ більшості практичних задач приходиться мати справу з наборами даних, представлених у виді векторів, таблиць, рядків або абстрактних множин з неупорядкованою структурою. Для розміщення перерахованих об'єктів у пам'яті ЕОМ апарат простих змінних виявляється явно недостатнім, приводячи до невиправданої громіздкості і логічної складності програм. Найбільш простою логічною структурою даних є обмежений вектор елементів фіксованого типу. Елементи вектора вважаються занумерованими послідовно відрізком ряду натуральних чисел і доступ до кожного з них може здійснюватися по його індивідуальному номеру. Простота подібних структур даних полягає в тому, що вони самим природним образом відображаються на лінійну структуру пам'яті ЕОМ.
Структурним аналогом векторів у мові Сі (як і в більшості інших мов програмування) є масиви змінних, тобто упорядковані послідовності елементів даних одного типу. З кожним таким масивом зв'язується його ім'я, тип елементів (а отже й обсяг машинної пам'яті, необхідної для розміщення одного елемента) і їхня кількість (тобто довжина масиву). Для опису масивів у Сі-програмах використовуються інструкції, подібні тим, що були введені для опису простих змінних. Розходження складається лише в тім, що тепер ці інструкції повинні містити інформацію про довжину масиву й у загальному випадку мають наступний формат:

<sc-specifier> type-specifier identifier[const-expression] <, ... >;

де sc-specifier є опис класу пам'яті (див. Лекцію 7, $ 2);
type-specifier - ім'я типу даних, якому належать елементи масиву;
identifier - ім'я змінної, що оголошена масивом;
const-expression - константний арифметичний вираз, що визначає кількість елементів у масиві (тобто його довжину). Наприклад, інструкції

```
int  f[10];
char  b[5];
```

визначають два масиви: масив f, що складається з десяти елементів типу int, і масив символів b, довжина якого дорівнює п'яти. Для звертання в програмі до індивідуального елемента будь-якого масиву варто вказати його ім'я і порядковий номер, який закладений в квадратні дужки:

```
c = 2*f[5] + 4;
```

причому останні в даному випадку розглядаються як символ операції взяття елемента, що має максимально високий пріоритет. Важливо знати, що компілятор мови Сі нумерує елементи масивів, починаючи з нуля (а не з одиниці, як це прийнято в багатьох мовах програмування). Тому в приведеному вище прикладі одним з операндів арифметичного вираження в правій частині є шостою (а не п'ятий!) елемент масиву f. Останній же елемент цього масиву повинний індексуватися числом 9. У загальному випадку порядкові номери елементів можуть задаватися довільними арифметичними виразами цілого типу, які будемо називати індексними виразами. Мова Сі не забезпечує можливості роботи з масивами змінних як з єдиним цілим: у якості операндів виражень можуть виступати лише окремі елементи. Однак посилання на ім'я масиву без наступного за ним індексу елемента все-таки мають визначений сенс, оскільки ототожнюється з адресою розміщення в пам'яті найпершого елемента цього масиву. Докладна мова про використання адрес і адресній арифметиці піде в наступній лекції.

$ 2. Рядки символів

У $ 3 попередній лекції, присвячена розглядові констант у мові Сі, ми вже познайомилися з поняттям символьного рядка як послідовності, яка складається з одного або більшого числа символів ASCII. Тепер нам треба обговорити питання про розміщення рядків у пам'яті ЕОМ і про їхнє використання в програмах. Справа в тім, що в мові Сі немає спеціального типу, який можна було б використовувати для опису рядків. Замість цього вони представляються у виді масивів елементів типу char. Таке рішення проблеми означає, що символи рядка розташовуються в сусідніх комірках пам'яті - по одному символі в

комірці. У той же час, настільки спрощене представлення про рядки трохи затрудняє практичну роботу з ними, оскільки в більшості випадків рядки цікавлять нас як цілісні в логічному відношенні одиниці інформації, а не як набори символів, з яких вони складаються. Частковий вихід із ситуації, що створилася, досягається шляхом вставки нуль-символу (Esc-послідовність \0) у кінець всякого рядка. Це означає, що під рядком варто розуміти послідовність символів, які починається в нульовому елементі масиву типу char і закінчуються символом \0. При цьому потрібно пам'ятати, що описуючи символьний масив необхідно за резервувати одну додаткову комірку пам'яті під збереження нуль-символу. Так, наприклад, для представлення рядка, що містить 40 символів, у програмі необхідно мати опис виду

```
char  string[41];
```

т.я. розмірність масиву повинна бути принаймні на одиницю більше істиної довжини рядка. Проте, через те, що в мові Сі немає засобів для роботи з масивами як з єдиним цілим,і не існує можливості маніпулювати рядками як неподільними одиницями інформації. Ця проблема знімається за рахунок використання функцій зі стандартної бібліотеки мови. Нижче приведені приклади деяких з них.

Ім'я функції і призначення: strcat - додавання рядка string2 у кінець рядка string1

Формат і опис аргументів:

```
char  *strcat(string1, string2)
char  *string1;      /*  Покажчик на рядок-приймач  */
char  *string2;      /*  Покажчик на рядок-джерело  */
```

Значення, що повертається, дорівнює адресі початку строки string1, тобто покажчикові на цей рядок.

Ім'я функції і призначення: strchr - пошук першого входження символу sym у рядок string

Формат і опис аргументів:

```
char  *strchr(string, sym)
const c har  *string;          /*  Покажчик на рядок  */
int        sym;             /*  Символ пошуку       */
```

Значення, що повертається, є покажчиком на перше входження символу sym у рядок string і дорівнює NULL, якщо цей символ у складі рядка не знайдений.
Ім'я функції і призначення: strcmp - порівняння рядків string1 і string2, вважаючи різними великі і малі букви латинського алфавіту

Формат і опис аргументів:

```
int  strcmp(string1, string2)
const char  *string1;   /*  Покажчик на перший рядок  */
const char  *string2;   /*  Покажчик на другий рядок  */
```

Значення, що повертається, дорівнює нулеві для ідентичних рядків і відмінно від нуля в противному випадку.
Ім'я функції і призначення: strcmpi - порівняння рядків string1 і string2, не роблячи розходження між великими і малими буквами латинського алфавіту
Формат і опис аргументів:

```
int  strcmpi(string1, string2)
const char  *string1;   /*  Покажчик на перший рядок  */
const char  *string2;   /*  Покажчик на другий рядок  */
```

Значення, що повертається, дорівнює нулеві для ідентичних рядків і відмінно від нуля в противному випадку.
Ім'я функції і призначення: strcpy - копіювання рядка string2 у рядок string1
Формат і опис аргументів:

```
char  *strcpy(string1, string2)
char  *string1;       /*  Покажчик на рядок-приймач  */
const char  *string2; /*  Покажчик на рядок-джерело  */
```

Значення, що повертається, дорівнює адресі початку рядка string1, тобто є покажчиком на цей рядок.
Ім'я функції і призначення: strlen - визначення довжини рядка string
Формат і опис аргументів:

```
int  strlen(string)
const char  *string;      /*  Покажчик на рядок  */
```

Значення, що повертається, дорівнює кількості символів у рядку string, крім завершального нуль-символу.
Ім'я функції і призначення: strlwr - заміна усіх великих букв латинського алфавіту в складі рядка string відповідними малими буквами
Формат і опис аргументів:

```
char  *strlwr(string)
char  *string;   /*  Покажчик на преутворений рядок  */
```

Значення, що повертається, дорівнює адресі початку рядка string, тобто покажчикові на цей рядок.
Ім'я функції і призначення: strstr - пошук першого входження подстроки string2 у рядок string1
Формат і опис аргументів:

```
char  *strstr(string1, string2)
const char  *string1;       /*  Покажчик на рядок     */
const char  *string2;       /*  Покажчик на підстроку  */
```

Значення, що повертається, є покажчик на перше входження підстроки string2 у рядок string1 і дорівнює NULL, якщо підстрока пошуку не виявлена в складі рядка string1.

Ім'я функції і призначення: strupr - заміна всіх малих букв латинського алфавіту в складі рядка string відповідними великими буквами
Формат і опис аргументів:

```
char  *strupr(string)
char  *string;   /*  Покажчик на преутворений рядок  */
```

Значення, що повертається, дорівнює адресі початку рядка string, тобто покажчикові на цей рядок.
Попередні описи всіх цих функцій (див. Лекцію 5, $ 3) поміщені у файл string.h і при їхньому використанні останній повинний бути включений до складу вихідного тексту програми за допомогою директиви препроцесора #include (див. Лекцію 7, $ 4 і приклади програм). Поняття покажчика, що зустрічається в цьому параграфі, буде докладно розглянуто в Лекції 4.

3. Ініціалізація змінних і масивів

Під ініціалізацією довільних елементів даних розуміється присвоєння їм деяких значень перед початком роботи програми. Це можна виконати, вводячи інціалізовані вирази безпосередньо в інструкції опису змінних. Для простих змінних інціалізатор має наступний формат:

= expression

де expression - довільне константне вираження. Наприклад:

```
char   sym = 0x73, slash = '/';
int    x = 10, y = 15;
  float  alpha = 4.7, pi = 3.1415, fi = pi/4;
```

У випадку ініціалізації масивів послідовність ініціалізованих виразів заключаються у фігурні дужки:
= {initializer-list}

де initializer-list - список константних виражень, розділених комами. Наприклад:

```
int    set[4] = { 3, 7, 9, 13 };
char   string[6] = { 's', 't', 'r', 'i', 'n', 'g' };
float  tab[10] = { 3.5, 6.7, 5.3 };
```

Помітимо, що ініціалізація масивів припустима лише при їхньому описі на зовнішньому рівні програми, т.я. поза тілом який би не було функції, або при явному призначенні класу пам'яті static. Більш детальна інформація з цього питання буде приведена в п'ятій і сьомій лекціях. Ті змінні або елементи масивів, для яких початкові значення явно не задані, ініціалізуються нулем при їхньому визначенні на зовнішньому рівні або при наявності опису класу пам'яті static, і залишаються невизначеними для автоматичних і реєстров змінних. Виключення є випадок, коли розмірність масиву не зазначена:

```
int  mas[] = { 1, 2, 3 };
```

У цьому прикладі довжина масиву визначається по числу інциалізованих значень. Ініціалізація символьних рядків може бути виконана в такий спосіб:

```
char  str[] = "hello";
```

що еквівалентно описові

```
char  str[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

У першому випадку довжина масиву дорівнює кількості символів у рядку плюс один, а в другому дорівнює кількості елементів у списку ініціалізації. З іншого боку, опис

```
char  line[80] = "This is a string";
```

оголошує масив, що складається з 80 елементів типу char, але лише 16 з них відмінні від нуль-символу. Використовуване в цьому параграфі поняття класів пам'яті буде докладно розглянуто в Лекції 7.

4. Керуючі конструкції мови Сі

Наприкінці попередньої лекції було введено фундаментальне поняття оператора, як основного конструкційного блоку програми на мові Сі. Саме оператори є закінченими інструкціями для виконуючої системи (комп'ютера). Ми познайомилися з найпростішими і найбільш важливими операторами, якими є будь-які вирази мови, що закінчуються крапкою з комою. Цю групу операторів прийнято називати виразами-операторами. У дійсному параграфі буде розглянута велика група операторів, які керують процесами виконання інструкцій програми. До них відносяться: умовний оператор, перемикач, кілька різновидів оператора циклу й оператори передачі керування. У літературі по мовах програмування оператори розглянутої групи часто називають керуючими конструкціями, тому що вони порушують природний лінійний порядок виконання розпоряджень програми.

4.1. Умовний оператор if-else

Умовний оператор є найпростішим і ефективним засобом, який дозволяє програмувати розгалужені алгоритми. Його формальний синтаксис задається наступною формулою:

```
if (expression)
statement1
else
```

statement2

причому else-частина не є обов'язковою і може бути опущена. Тут if і else - ключові слова мови Сі; expression - довільний вираз, який приводиться до логічного значення; statement1 і statement2 - прості або складені оператори.
Семантика. Насамперед обчислюється значення виразу, що стоїть в дужках, і отриманий результат порівнюється з нулем, який відповідає його інтерпретації в термінах "істина"/"неправда". Якщо результатом є логічне значення "істина" (вираження відмінне від нуля), то виконуються дії, що відповідають statement1. У противному випадку здійснюється перехід до statement2 у else-частині умовного оператора, а при її відсутності - до оператора, що безпосередньо знаходиться за оператором if. У приведеному нижче прикладі

```
f (i > 0)
y = x/i;
else
{ x = i/2; y = x + 4; }
```

оператор y = x/i; буде виконаний, якщо значення і більше нуля. У противному випадку виконується група операторів x = i/2; y = x + 4; Зауваження. У конструкціях виду

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

де один оператор if вкладений в інший, else-частина зв'язується з найближчим попереднім оператором if. Щоб змінити ця угода, необхідно використовувати фігурні дужки:

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

4.2. Оператор-перемикач switch

Перемикач є спеціальним випадком умовного оператора і дозволяє здійснювати різноманітний вибір, заміняючи групу вкладених операторів if-else. У загальному випадку від має наступний формат:

```
switch (expression)
{
 case  const-expression:  statements
  .          .          .
  .          .          .
 default:  statements
  .          .          .
  .          .          .
 case  const-expression:  statements
}
```

де будь-яка case- або default-частина може її небути , а також можуть бути опущені statements у кожній з цих частин. Тут switch, case і default - ключові слова мови Сі; expression - довільне вираз цілого типу; const-expressi- on - константний вираз, яку обробляється в період компіляції програми; statements - група операторів. Семантика. Попередньо обчислене значення виразу в круглих дужках порівнюється з const-expression у всіх варіантах case і керування передається тій групі операторів statements, що відповідає знайденому значенню. Коли значення жодного з константних виражень не збіглося зі

значенням вираження expression, виконуються оператори, зв'язані з міткою default, а при її відсутності - оператор, який безпосередньо слідує за оператором switch. Ніякі два константних вираза в одному операторі-перемикачі не можуть мати однакових значень. У наступному прикладі значення змінної operation, що має тип char, визначає ту арифметичну операцію, що повинна бути виконана над змінними x і y:

```
switch (operation)
  {
   case '+':
          z = x + y;
          break;
   case '-':
          z = x - y;
          break;
   case '*':
          z = x*y;
          break;
   case '/':
          z = x/y;
  }
```

Зауваження. Ключове слово case разом з const-expression служать просто міткою відповідних операторів, і якщо будуть виконуватися оператори для деякого варіанта case, то далі будуть виконуватися оператори всіх наступних варіантів доти, поки не зустрінеться оператор break (див. нижче). Тому якщо декільком різним значенням вираза expression повинні відповідати ті самі дії, можна скористатися наступною конструкцією:

```
switch (key)
  {
   case '0':
   case '1':
   case '2':
   case '3':
   case '4':
   case '5':
   case '6':
   case '7':
   case '8':
   case '9':
          printf ("Це цифра");
  }
```

4.3. Оператор циклу while

Оператор while дозволяє циклічно виконувати визначену послідовність операторів доти, поки істинно деяка умова, що перевіряється перед початком чергової ітерації циклу (цикл із передумовою).Формально його синтаксис може бути описаний наступною формулою:

```
while (expression)
  statement
```

де while - ключове слово мови Сі; expression - довільне вираз, який приводиться до логічного значення (умова продовження ітерацій); statement - простий або складений оператор (тіло циклу). Семантика. Якщо значення виразу expression істинно (відмінно від нуля), то тіло циклу виконується доти, поки це вираження не стане помилковим (рівним нулеві), причому перевірка виконується кожний раз перед початком чергової ітерації циклу. Якщо ж значення виразу неправда в момент ініціалізації циклу, то

тіло циклу не виконується жодного разу і керування передається операторові, що слідує за оператором while.
Як приклад використання оператора циклу з передумовою, розглянемо фрагмент програми, яка визначає довжину рядка символів string:

```
count = 0;
while (string[count] != '\0')
  count++;
```

Після закінчення роботи цього фрагмента, значення змінної count дорівнює кількості символів у рядку string, не рахуючи завершальний нуль-символ.

4.4. Оператор циклу do-while

Принципове призначення цього оператора таке ж, як і розглянутого в п. 1.3. оператора while. Семантичне розходження двох цих операторів циклу складається лише в тім, що в розглянутому випадку умова продовження перевіряється після завершення чергової ітерації (цикл з пост умовою). У загальному виді оператор do-while записується в так:

```
do
  statement
while (expression);
```

де do і while - ключові слова мови Сі; statement - простий або складений оператор (тіло циклу); expression - довільний вираз, який приводиться до логічного значення (умова продовження ітерацій). Семантика. Після однократного виконання тіла циклу обчислюється значення виразу expression. Якщо знайдене значення істинне (відмінно від нуля), то виконується чергова ітерація циклу. Цей процес повторюється доти, поки значення виразу не стане помилковим (рівним нулеві), після чого керування передається операторові, що безпосередньо слідує за оператором do-while. Таким чином, тіло циклу завжди буде виконане хоча б один раз. Як приклад використання циклу з пост умовою, приведемо фрагмент програми, що виконує копіювання рядка символів string1 у рядок string2:

```
count = 0;
do
  string2[count] = string1[count];
while (string1[count++]);
```

4.5. Оператор циклу for

Оператор for є найбільш могутнім засобом реалізації циклічних алгоритмів і керування циклічними процесами. Він поєднує в собі ініціалізацію змінних, перевірку умови продовження ітерацій і модифікацію змінних перед виконанням чергової ітерації, яка по своїй суті еквівалентно наступній групі операторів:

```
expression1
while (expression2)
  { statement
    expression3; }
```

Загальна форма запису оператора циклу for така:

```
for (expression1; expression2; expression3)
  statement
```

де for - ключове слово мови Сі; expression1, expression2 і expression3 - довільні вирази, причому друге з них приводиться до логічного значення; statement - простий або складений оператор (тіло циклу). Кожне з трьох виражень, що входять у заголовок циклу, може бути опущене при збереженні

наступної за ним крапки з коми. Семантика. Перед початком виконання циклу обчислюється значення ініціалізованого виражу expression1 і значення виразу expression2, грає роль умови продовження ітерацій. Якщо expression2 істинно (відмінно від нуля), то виконується оператор statement тіла циклу, обчислюється коригувальний вираз expression3 і знову перевіряється істинність вираза expression2.Цей процес повторюється доти, поки умова продовження ітерацій не стане помилковим (рівним нулеві), після чого керування передається операторові, що слідує за оператором for. У випадку відсутності виразу expression2 йому умовно присвоюється постійне значення "істина", яке рівносильне нескінченному циклічному процесові. У наступному програмному фрагменті, що обчислює суму елементів числового масиву, оператор циклу for використовується для організації послідовного доступу до всіх елементів цього масиву:

```
s = 0;
for (i = 0; i < n; i++)
  s = s + a[i];
```

Кожний з виразів у заголовку циклу може складатися з декількох підвиразів, зв'язаних між собою операцією кома (,). Ця операція забезпечує послідовне обчислення своїх операндів з лів на прво, причому її результатом є значення правого операнда. Стосовно до оператора циклу for, операція кома найчастіше використовується в тих випадках, коли необхідно мати декілька ініціалізованих і коригувальних виразів. Цю ситуацію можна проілюструвати наступним програмним фрагментом, що виконує інвертування послідовності з n елементів:

```
for (i = 0, j = n-1; i < j; i++, j--)
  { buf = a[i]; a[i] = a[j]; a[j] = buf; }
```

4.6. Оператор закінчення break

В ряді випадків виникає необхідність завершити виконання якого-небудь оператора циклу або оператора-перемикача раніш, ніж це відбудеться відповідно до прийнятої семантики цих операторів. Для цієї мети служить оператор break, що припиняє виконання найближчого вкладеного оператора while, do-while, for або switch і передаэ керування наступному операторові. Ідентифікатор break є ключовим словом мови Сі. Нехай, наприклад, необхідно знайти суму елементів числового масиву до першого негативного елемента. Цю задачу можна вирішити за допомогою наступного програмного фрагмента:

```
s = 0;
for (i = 0; i < n; i++)
  if (a[i] >= 0)
    s = s + a[i];
  else
     break;
```

Помітимо, що поява оператора break поза одного з операторів while, do-while, for або switch призводить до помилки на етапі компіляції програми.

4.7. Оператор продовження continue

При виникненні необхідності припинити виконання поточної ітерації найближчого зовнішнього оператора while, do-while або for і перейти до початку наступної ітерації, може бути використаний оператор continue. У випадку операторів while і do-while чергова ітерація починається з обчислення і перевірки істинності виразу expression (див. пп. 1.3. і 1.4.), а в операторі for - з обчислення значення коригувального вираза expression3 (див. п. 1.5.). Ідентифікатор continue є ключовим словом мови Сі.

Нехай у прикладі п 1.6. тепер необхідно знайти суму всіх позитивних елементів числового масиву. Використовуючи оператор continue, це можна зробити в такий спосіб:

```
s = 0;
for (i = 0; i < n; i++)
  { if (a[i] < 0)
      continue;
    s = s + a[i]; }
```

4.8. Оператор переходу goto

Як і в переважній більшості мов програмування, у мові Сі мається можливість безумовної передачі керування будь-якому операторові в межах поточної функції, хоча гнучкість керуючих конструкцій цієї мови дозволяє майже завжди обходитися без такої можливості. Формальний систаксис оператора переходу описується наступною формулою:

```
goto label;
```

де goto - ключове слово мови Сі, а label - мітка оператора в межах поточної функції (формальне визначення мітки дане в наступному абзаці).Відповідно до синтаксичних правил мови Сі, кожному операторові програми може бути присвоєне унікальне ім'я, називане міткою оператора. Сама мітка є правильним ідентифікатором мови, за яким слідує символ двокрапка (:). Мітки операторів можуть бути використані тільки лише як об'єкти посилань в операторі goto. В всіх інших випадках вони ігноруються компілятором. Як приклад використання оператора goto для виходу з вкладених керуючих операторів, розглянемо задачу знаходження першого елемента двовимірного масиву, рівного заданому числу x:

```
x = 17;
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    if (a[i][j] == x)
      goto L01;
/*  Дії, виконувані у випадку,
    якщо елемент x не знайдений      */
.....
.....
.....
L01: .....
```

4.9. Оператор повернення return

Оператор return припиняє виконання поточної функції і повертає керування тієї функції, з якої була викликана поточна. У загальному випадку він має наступний формат:

```
return expression;
```

де return - ключове слово мови Сі, а expression - довільне необов'язковий вираз. Семантика. Після припинення виконання поточної функції і передачі керування функції, що викликала, виконання останньої продовжується з оператора, який безпосередньо слідує за крапкою виклику. Крім цього, при наявності виразу expression його значення передається (можливо після попереднього перетворення типу) функції, що викликала.

Якщо ж вираз в операторі return відсутній, то значення, що повертається поточною функцією, вважається невизначеним. Зауваження. У випадку використання оператора return у головній функції (тобто функції, що має ім'я main), його дія приводить до припинення виконання програми і передачі керування операційній системі. Більш докладно цей оператор буде розглянутий у Лекції 5, присвяченої функціям у мові Сі.

Лекція 4

Адреси і покажчики. Операції одержання адреси і непрямої адресації. Ототожнення масивів і покажчиків. Адресна арифметика. Покажчики на масиви. Масиви покажчиків і багатомірні масиви. Динамічне виділення пам'яті під масиви. Ініціалізація покажчиків.

1. Адреси і покажчики

Під час виконання будь-якої програми, використовувані нею дані розміщаються в оперативній пам'яті ЕОМ, причому кожному елементові даних ставиться у відповідність його індивідуальна адреса. При реалізації багатьох алгоритмів і представленні складних логічних структур даних часто виявляється можливість безпосередньої роботи з адресами пам'яті. Подібна ситуація виникає, наприклад, при обробці масивів змінних. Дійсно, оскільки сусідні елементи масиву розташовуються в суміжних комірках пам'яті, то для переходу від одного елемента до іншого можна замість зміни значення індексного вираження маніпулювати адресами цих елементів. Припустимо для визначеності, що нульовий елемент цілочислового масиву розташований у комірці пам'яті з номером Ao. Тоді, знаючи, що довжина елемента даних типу int складає два байти, неважко обчислити номер комірки, в якій буде знаходитися i-ий елемент цього масиву:

Ai = Ao + 2*i

На перший погляд робота з адресами може показатися стомлюючою і марною. Насправді ж вона є навіть більш ефективною, ніж робота з індексами, оскільки в процесі компіляції програми будь-яке індексне вираження трансформується в операції над адресами. Об'єкти мови Св, значеннями яких є адреси оперативної пам'яті, одержали назву покажчиків. У загальному випадку покажчики є змінними величинами і над ними можна виконувати визначений набір операцій подібно тому, як ми оперували звичайними числовими змінними. У мові Сі будь-який покажчик має базовий тип, який збігається з типом елемента даних, на який може посилатися цей покажчик. Така умова, можливо обмежувальне, істотно спрощує і робить значно більш ефективну роботу з покажчиками. Переміні-покажчики, як і змінні будь-яких інших типів, перед їхнім використанням у програмі повинні бути попередньо оголошені в одній з інструкцій опису даних. У випадку покажчиків на прості змінні це робиться в такий спосіб:

<sc-specifier> type-specifier *identifier <, ... >;

Тут type-specifier задає тип змінної, на яку посилається покажчик з ім'ям identifier, а символ зірочка (*) визначає саму змінну як покажчик. Опис класу пам'яті sc-specifier буде докладно розглянутий у Лекції 7. Приведемо кілька прикладів правильного опису покажчиків у програмі:

```
int    *ptr;
long   *sum;
float  *result, *value;
```

Кожна з цих інструкцій говорить про те, що відповідна змінна є покажчик на елемент даних визначеного типу, а комбінація, наприклад, виду *ptr являє собою величину типу int. Власне кажучи це означає, що подібні комбінації можуть використовуватися як операнди довільних виражень. Зокрема, зберігаючи позначення попереднього приклада, ми могли б написати

```
*sum = 0;
for (*ptr = 1; *ptr <= 100; (*ptr)++)
   *sum = *sum + (*ptr)*(*ptr);
```

що відповідає фрагментові програми обчислення суми квадратів перших ста натуральних чисел. Круглі дужки в коригувальному виразу оператора циклу є істотними. Зауваження. Незважаючи на те, що будь-який покажчик у мові Сі має прив'язку до конкретного типу даних, існує можливість в окремих випадках створювати покажчики невизначеного типу. Це робиться за допомогою ключового слова void, використовуваного замість імені типу в інструкції опису цього покажчика. Так, наприклад, інструкція

```
void  *poiter;
```

визначає змінну poiter як покажчик без конкретного посилання на його базовий тип. Однак подібні описи не є типовими і можуть бути використані лише при оголошенні формальних параметрів функцій і визначенні типу поверта тією або іншою функцією значення (див. Лекцію 5, $ 1). Строго говорячи, компілятор мови Сі розглядає комбінації виду

*identifier

в складі виразів як деяку операцію над покажчиками. Ця операція, символом якої саме і є зірочка перед ім'ям покажчика, зветься операції непрямої адресації і служить для доступу до значення, розташованому по заданій адресі. Існує й інша операція, у визначеному змісті протилежна операції непрямої адресації й названа операцією одержання адреси. Вона позначається символом амперсенда (&) перед ім'ям простої змінної або елемента масиву:

&identifier або &identifier[expression]

і зіставляє своєму аргументові адреса його розміщення в пам'яті, тобто покажчик. Природно, що цим аргументом може бути і покажчик, оскільки покажчики, як і інших змінні, зберігаються в осередках оперативної пам'яті. Усілякі вираження, побудовані з використанням покажчиків або операторів * і &, прийнято називати адресними вираженнями, а самі арифметичні операції над покажчиками - адресною арифметикою. Одномісні операції * і & мають такий же високий пріоритет, як і інші унарные операції, і в складі виражень обробляються праворуч ліворуч. Саме з цієї причини ми звернули увагу на необхідність круглих дужок у вираженні (*ptr)++ попереднього приклада, тому що без них оператор ++ відносився б до покажчика ptr, а не до значення, на яке посилається цей покажчик.
Зауваження. Якщо, наприклад, mas є масив змінних, то вираження &mas[0] рівносильне простому вживанню імені масиву без наступних за ним індексного вираження, оскільки останнє ототожнюється з адресою розміщення в пам'яті найпершого елемента цього масиву. От кілька прикладів використання покажчиків і адресних виражень.

1. Аргументами функції форматированного введення scanf є адреси змінних, котрим повинні бути привласнені прочитані значення:

```
scanf("%d %d", &m, &n);
```

2. Наступна пара операторів

```
px = &x;
y  = *px;
```

де змінна px оголошена попередньо як покажчик, рівносильна безпосередньому присвоюванню

```
y = x;
```

3. При виконанні наступного фрагмента програми порівняння в операторі if завжди буде істинно, оскільки значення покажчика numptr збігається з адресою змінної number:

```
int  number;
int  *numptr = &number;
scanf("%d %d", &number, numptr);
if (number == *numptr)
  printf("Порівняння щире");
else
  printf("Порівняння помилкове");
```

2. Ототожнення масивів і покажчиків.Адресна арифметика

У цьому параграфі нам треба розібратися в тому, який зміст варто вкладати в арифметичні операції над покажчиками й у якому відношенні між собою знаходяться масиви і покажчики. Підставою для обговорення останнього питання є відзначений вище факт, що при відсутності індексного вираження ім'я масиву власне кажучи є покажчик на його нульовий елемент. Тому доступ до і-ого елемента цього масиву можна одержати, збільшуючи значення покажчика на відповідну величину. Розглянемо як приклад наступний опис

```
int  a[10];
```

визначальний масив з десяти елементів типу int. Оскільки a == &a[0], те адреса елемента a[i] дорівнює

a + sizeof(int)*i

Хоча приведений запис і покаже суть справи, птим не менш вона є не досить зручною через свою громіздкість. Дійсно, враховуючи те що кожний елемент масиву a має тип int і займає sizeof(int) байт пам'яті, з адресного виразу можна було б виключити інформацію про довжину елемента масиву. Для цього досить, наприклад, прийняти согласованість про те, що вираз виду a+i саме і визначає адреса i-ого елемента, тобто

&a[i] == a+i

Тоді позначення a[i] стає еквівалентним адресному виразу *(a+i) в тому змісті, що обоє вони визначають те саме числове значення, а саме:

a[i] == *(a+i)

Нехай теперє пара описів

```
int  a[10];
int  *pa;
```

Виконуючи операцію присвоювання

pa = a або pa = &a[0]

ми встановлюємо покажчик pa на нульовий елемент масиву a і тому справедливі рівності

&a[i] == pa+i і a[i] == *(pa+i)

т. я. операцію pa+i, що збільшує значення покажчика, можна інтерпретувати як зсув вправо на i елементів базового типу. Все це означає, що кожне звертання до i-ого елемента масиву або його адресі припустимо представляти як в індексній формі, так і мовою покажчиків.Звернему увага на одну важливу обставину, яке відрізняє масиви від покажчиків. Оскільки останні є змінними величинами, то виявляються припустимими наступні адресні вирази

pa = pa+i або pa = a або pa++

Однак через те, що ім'я масиву є константа, що визначає фіксовану адресу розміщення цього масиву в пам'яті ЕОМ, операції виду

a = pa або a = a+i або a++ або pa = &a

варто вважати позбавленими якого-небудь змісту. Продовжуючи далі аналогію масивів і покажчиків, необхідно дозволити індексування покажчиків,

pa[i] == *(pa+i) або &pa[i] == pa+i

що є зовсім природним, якщо позначення pa[i] розуміти як взяття значення за адресою pa+i. Індексуючи елементи масиву, ми по суті знаходимося в рамках тієї ж самої домовленості. Помітимо, що було б грубою помилкою вважати, що опис

```
int  a[10];
int  *pa;
```

цілком рівносильні одне одномуу. Справа в тім, що в першому випадку визначена адреса початку масиву і виділене місце в пам'яті ЕОМ, достатня для збереження десяти елементів. В іншому ж випадку покажчик має невизначене (нульове) значення і не посилається ні на який зв'язний ланцюжок байт. Таке значення покажчика звичайно позначається символічним ім'ям NULL, визначеним у стандартному файлі stdio.h. Для того, щоб покажчик став цілком еквівалентний масивові, необхідно змусити його посилатися на область пам'яті відповідної довжини. Це можна зробити за допомогою стандартних функцій malloc() і alloca() (див. $ 4), що захоплюють необхідну кількість байт пам'яті і повертають адресу першого з них. Так, наприклад, після виконання оператора

pa = (int*)malloc(10*sizeof(int));

визначений вище масив a і покажчик pa стають у повному змісті еквівалентними. Однак друге рішення буде більш гнучким, тому що тут викликана пам'ять виділяється динамічно в процесі виконання програми і може бути при необхідності повернута системі за допомогою функції free() (див. $ 4 дійсної лекції), чого не можна зробити у випадку масиву. Зупинимося особливо на питанні використання покажчиків для представлення й обробки символьних рядків.
Оскільки в мові Си немає спеціального типу даних, який можна було б використовувати для опису символьних рядків,останні зберігаються в пам'яті ЕОМ у виді масивів символів. Так, наприклад, опис

char string[] = "Це рядок символів";

визначає масив двадцяти елементів типу char, визначая їх символами рядка. Звертання до якого-небудь елемента цього масиву забезпечує доступ до окремого символу, а адреса початку рядка дорівнює &string[0]. З іншого боку, через те, що строкова константа в правій частині нашого опису ототожнюється компілятором з адресою її першого символу, правильний є запис наступного виду:

char *strptr = "Це рядок символів";

визначая покажчик значенням адреси рядка-константи. Розходження двох приведених описів таке ж, як і відзначене вище розходження масивів і покажчиків.Так, в другому випадку ми могли б написати

strptr = strptr + 4;

змістивши тим самим покажчик на початок другого слова рядка. Більш того, є припустимим присвоювання

strptr = "Це інший рядок символів";

значення покажчика, що змінює, (але не виконуэкопіювання символів рядка!). Подібна операція не має змісту для імені масиву, яке у внутрішнім машинному представленні ототожнюється з фіксованою адресою його нульового елемента. Крім визначеної вище операції збільшення покажчика, можна також зазначити операцію його зменшення, яка рівносильно рухові вздовж масиву в напрямку зменшення значень індексу. Більш того, безліч значень змінних-покажчика є упорядкованим (тому що упорядковані адреси оперативної пам'яті) і тому використання покажчиків у якості операндов умовних і логічних виражень не суперечить семантичним правилам мови Си.

3. Покажчики на масиви. масиви покажчиків і багатомірні масиви

Введене в попередньому параграфі поняття покажчика на просту змінну природним образом поширюється на будь-які структуровані типи даних. Зокрема, декларація

float (*vector)[15];

визначає ім'я vector як покажчик на масив п'ятнадцяти елементів типу float, причому круглі дужки в цьому записі є істотними. Звертання до і-ого елемента такого масиву буде виглядати в такий спосіб:

(*vector)[i]

Визначаючи покажчик на масив, ми зберігаємо всі переваги роботи з покажчиками і, крім того, вимагаємо від компілятора виділити реальну пам'ять для розміщення елементів цього масиву. Тому що самі по собі покажчики є змінними, та неважко побудувати обмежений вектор елементів-покажчиків на деякий базовий тип даних. Такі структури даних у мові Сі прийнято називати масивами покажчиків. Їхній опис будується на тій же синтаксичній основі, що й описзвичайних масивів. Наприклад, інструкція

char *text[300];

визначає масив трьохсот покажчиків на елементах типу char. Оскільки кожен окремий елемент цього масиву може зберігати адресу початку деякого ланцюжка символів, то після фактичного виділення пам'яті під розміщення трьохсот таких ланцюжків і присвоєння адреси кожної з них визначеному елементові масиву, весь масив покажчиків буде задавати набір відповідної кількості рядків змінної,

взагалі говорячи, довжини. Елементи масиву покажчиків можуть бути описані подібно тому, як описані окремі покажчики і звичайні масиви:

```
char  *week[] = { "Понедєльник",
                  "Вівторок",
                  "Середа",
                  "Четвер",
                  "П'ятниця",
                  "Субота",
                  "Неділя" };
```

Згадуючи приведену аналогію між масивами і покажчиками, можна сказати, що масив покажчиків у визначеному змісті еквівалентний "масивові масивів", який у загальному виді варто було б описувати в такий спосіб:

```
type-specifier identifier[const-expression][const-expression];
```

де всі позначення використані в тім же змісті, що і раніше. Так, опис

```
char  table[10][20];
```

визначає масив десяти масивів, кожний з яких містить по двадцятьох елементів типу char. Легко помітити, що це є не що інше, як синонім двовимірного масиву, причому перший індекс визначає номер рядка, а другий - номер стовпця. Очевидно, що бажаючи зберегти тісний зв'язок масивів і покажчиків, варто зажадати, щоб двовимірні масиви розміщалися в пам'яті ЕОМ по рядках, ототожнивши ім'я масиву з адресним посиланням &table[0][0]. Звертання ж до індивідуальних елементів двовимірного масиву здійснюється, як і у випадку одного вимірного, за допомогою індексних виразів. Відмінність масиву покажчиків від масиву масивів складається, головним чином, в тому, що в першому випадку резервуються лише комірки пам'яті для збереження адрес рядків двовимірної таблиці, у той час як реальна пам'ять під розміщення елементів кожного рядка не виділяється. В другому ж випадку цілком визначений обсяг пам'яті, займаною всією таблицею. З іншого боку, загальна подібність між двома цими структурами даних дозволяє працювати з масивами покажчиків точно так само, як і з двовимірними масивами, використовуючи, наприклад, подвійну індексацію

```
week[2][3]
```

для виділення четвертого по рахунку символу в третьому рядку, і навпаки, розглядаючи посилання виду

```
table[i]
```

як адреса нульового елемента і-го рядка таблиці table. Така можливість виглядає досить природним, якщо замість терміна "багатомірний масив" завжди використовувати поняття "масив масивів". Приведена аналогія між масивами покажчиків і масивами масивів дає можливість додати цілком конкретний зміст виразу виду

```
table[i] + k
```

задаючи адресу k-го елемента і-го рядка масиву table, що у термінах операції взяття адреси визначається як

```
&table[i][k]
```

Тому поряд із традиційним посиланням

```
table[i][k]
```

призначення елемента (i, k) цього масиву можна користуватися еквівалентної їй посиланням

```
*(table[i] + k)
```

мовою покажчиків. Далі згадуючи, що ім'я будь-якого масиву при відсутності індексних виразів ототожнюється з адресою його найпершого елемента, неважко зрозуміти, що вираз

```
table + j
```

є звичайним адресним виразом, що визначає розміщення в пам'яті нульового елемента j-го рядка таблиці table. Неважко помітити, що незважаючи на спільність властивостей, масиви покажчиків забезпечують можливість більш гнучкого маніпулювання даними, ніж багатомірними масивами. Подальше збільшення гнучкості структур даних зв'язано з поняттям непрямого покажчика або " покажчика на покажчик", який може бути визначений в такий спосіб:

<sc_specifier> type-specifier **identifier <, ... >;

Тут знову зберігається аналогія з розглянутими вище об'єктами, тобто такий опис виявиться цілком рівносильним двовимірному масивові після того, як буде виділена реальна пам'ять під збереження адрес його рядків і розміщення елементів кожного окремого рядка. Це можна зробити, використовуючи, наприклад, функцію malloc() або alloca() (див. $ 4):

```
double  **dataptr;
dataptr = (double**)alloca(m*sizeof(double*));
for (i = 0; i < m; i++)
  dataptr[i] = (double*)alloca(n*sizeof(double));
```

В останньому прикладі здійснюється розміщення в пам'яті ЕОМ двовимірного масиву розміру m*n елементів типу double.
Продовжуючи побудову, можна було б по індукції ввести поняття масиву довільного числа виміру і покажчика будь-якого рівня косвенности, маючи встановлену вище еквівалентність між цими об'єктами в одномірному і двовимірному випадках. Однак, з огляду на порівняно виняткове практичне використання таких структур даних і в той же час логічну простоту їхньої побудови, ми не будемо особливо зупинятися на цьому питанні.

4. Динамічне виділення пам'яті під масиви

У двох попередніх параграфах під час обговорення питання про еквівалентності масивів і покажчиків ми скористалися стандартними функціями malloc() і alloca() для динамічного виділення пам'яті під збереження елементів масиву. Тут будуть розглянуті деякі деталі цієї проблеми. У багатьох задачах обчислювальної математики і при реалізації алгоритмів обробки інформаційних структур виникає потреба роботи з масивами, кількість елементів яких змінюється від одного прогону програми до іншого. Найпростіше рішення цієї проблеми складається в статичному описі відповідних масивів з вказівкою максимально необхідної кількості елементів. Однак такий підхід призводить, як правило, до невиправданого завищення обсягу пам'яті, необхідної для роботи програми. Альтернативне рішення полягає з використанням покажчиків для представлення масивів змінних. Нехай нам необхідно написати програму скалярного множення векторів A і B, розмірність яких заздалегідь не відома. Для цього запишемо в такий спосіб. Опишемо в заголовку програми змінну m, що визначає довжину відповідних масивів, і покажчики a, b, c, які будуть визначати розміщення в пам'яті векторів-співмножників і вектора-результату:

```
int    m;
float  *a, *b, *c;
```

Після того, як значення m буде визначене (воно може бути, наприклад, введено з клавіатури термінала), необхідно виділити достатній обсяг пам'яті для збереження всіх трьох векторів. Оскільки мова тут йде про динамічне розміщення масивів у процесі виконання програми, ми повинні скористатися однією з трьох спеціальних функцій, що входять до складу стандартної бібліотеки і зведення до якої приведені нижче.
Ім'я функції і призначення: alloca - резервує size байт пам'яті з ресурсу програмного стека; виділена пам'ять звільняється по завершенні роботи поточного програмного компонента (див. Лекцію 7)

Формат і опис аргументів:

```
void  *alloca(int size) /*  Необхідна кількість байт пам'яті  */
```

Значення, що повертається, є покажчиком типу char на перший байт зарезервованої області програмного стека і дорівнює NULL при відсутності можливості виділити пам'ять необхідного розміру. Для

одержання покажчика на тип даних, відмінний від char, необхідно застосувати до значення, що повертається, операцію явного перетворення типу (див. Лекцію 2, $ 6).
Ім'я функції і призначення: calloc - резервує пам'ять для розміщення n елементів масиву, кожний з яких має довжину size байт, виводячи всі елементи нулями; виділена пам'ять звільняється по завершенні роботи програми або за допомогою функції free() (див. нижче)

Формат і опис аргументів:

```
        void  *calloc(int n, int size)
  /* n - Загальна кількість елементів у масиві  */
  /* size - Довжина в байтах кожного елемента       */
```

Лекція 5

Функції в мові Сі. Формальні і фактичні параметри. Механізм передачі параметрів. Значення, що повертаються. Використання покажчиків як аргуменів функцій. Попередній опис функцій. Аргументи командного рядка.

1. Функції в мові Сі. Формальні і фактичні параметри. Механізм передачі параметрів.

Значення, що повертаються.

Будь-яка програма, написана мовою Сі представляє собою сукупність функцій, що виконують основну роботу з реалізації деякого алгоритму. Кожна з них, у свою чергу, є незалежний набір описів і операторів, що знаходиться між заголовком функції і її кінцем. Всі об'єкти, визначені в тілі функції, обмеженому відкриваючими і закриваючими фігурними дужками, є локальними для цієї функції в змісті області видимості і часу існування. Деталі цих понять будуть розібрані в Лекції 7. У складі загальної програми будь-яка функція ідентифікується своїм власним унікальним ім'ям, яким може бути будь-яке правильне ім'я в розумінні граматики мови Сі (див. Лекцію 1, $ 3). Та функція, з якої починається виконання програми, називається *головною функцією* і повинна мати визначене ім'я main. Всі інші функції, що входять у програму, запускаються в роботу шляхом їх прямого чи опосередкованого (через інші функції) виклику з головної функції. Функції грають надзвичайно важливу роль при підготовці Сі-програми. Дійсно, використовуючи функції, вихідну задачу можна представити у виді послідовності більш простих задач, кожна з який реалізує деяку частину загального алгоритму. Такий підхід до розробки програмного продукту іноді називають *методом декомпозиції*. Оперуючи функціями, що складаються з порівняно невеликого числа операторів, значно легше задовольнити вимоги структурного програмування і знизити трудозатрати при налагодженні програмного забезпечення. З іншого боку, у вигляді функцій можуть бути реалізовані окремі часто використовувані алгоритми. Включення таких функцій до складу стандартних бібліотек дає принципову можливість не перепрограмовувати всякий раз найбільш ходові методи, алгоритми й операції. Для організації зв'язку між незалежними функціями в мові Сі використовується або апарат формальних/фактичних параметрів, або набір глобальних чи зовнішніх змінних. *Формальними параметрами* ми будемо називати аргументи функції, що знаходяться у її заголовку й імена яких використовуються для побудови тіла функції при її визначенні. Вони можуть мати будь-який простий чи структурований тип, підтримуваний наявним компілятором мови чи визначений у програмі за допомогою інструкції typedef (див. Лекцію 8, $ 8). *Фактичними параметрами* є довільні вирази, значення яких передаються формальним параметрам при зверненні до функції. У такий спосіб реалізується можливість передачі необхідної інформації від викликаючої функції до функції, до якої звертаються, безпосередньо в момент її виклику. У свою чергу, ім'я функції, яку викликають може служити носієм вихідного значення, одержуваного в результаті роботи цієї функції, роблячи його доступним функції, що викликає. Випадок, коли результатом роботи функції є сукупність значень, буде розглянутий у $ 2 цієї лекції. Обговорення ж питання про використання глобальних об'єктів ми поки відкладемо до Лекції 7. Згадування про те, що ім'я функції може бути використане для передачі виробленого цією функцією значення, підказує необхідність зв'язати з цим ім'ям конкретний тип даних з числа підтримуваних компілятором мови Сі. Цей зв'язок встановлюється при визначенні самої функції

чи при складанні її попереднього опису (див. $ 3 цієї лекції). Визначаючи функцію у відповідному програмному компоненті, потрібно, насамперед , указати її ім'я і тип значення, що повертається, задати список формальних параметрів і визначити тип кожного з них. Таку сукупність описів прийнято називати *заголовком функції.* Слідом за ним повинне розміщуватися *тіло функції*, що представляє собою правильний блок, тобто набір описів і операторів, вставлених у фігурні дужки. У мові Сі визначення функції має наступний формат:

```
<sc-specifier> <type-specifier> declarator (<parameter-list>)
<parameter-declarations>
function-body
```

Тут sc-specifier є описувач класу пам'яті (static чи extern), докладно розглянутий у Лекції 7. type-specifier і declarator разом визначають тип значення, що повертається функцією, і її ім'я, причому відсутність першого з них рівносильно типу int. У приведеній схемі, declarator найчастіше є просто ідентифікатором функції, перед яким може стояти символ зірочка (*), якщо ця функція повертає покажчик на елемент даних відповідного типу. Однак у ряді випадків він може мати і більш складний формат. Ніяка функція не повинна повертати масивів чи інших функцій, але припустима передача покажчиків на ці об'єкти. У тих випадках, коли функція не виробляє ніякого значення чи повертає покажчик невизначеного типу (див. Лекцію 4, $ 1), її опис повинний починатися з ключового слова void, що знаходиться на місці імені типу значення, що повертається. Наприклад,

```
void  work(n, beta)
int    n;
float  beta;
{ .................
  .................
  .................
}
```

Parameter-list являє собою необов'язковий список формальних параметрів (аргументів) обумовленої функції, розділених комами:

```
(<identifier<, identifier> ... >)
```

де identifier – суть ім'я формального параметра. Описи формальних параметрів parameter-declarations у заголовку функції мають той же самий формат, що і розглянуті раніше описи простих і структурованих змінних, і служать для визначення типу цих параметрів. Ті параметри, імена яких не оголошені явно в одній з інструкцій опису, по умовчанню одержують тип int, що призначається компілятором. Пам'ять під розміщення формальних параметрів виділяється динамічно з ресурсу програмного стека в момент звертання до відповідної функції. Це означає, що параметри завжди повинні мати клас пам'яті auto чи register (див. Лекцію 7, $2), причому перший з них призначається компілятором за умовчанням. Тіло функції, будучи правильним блоком чи складеним оператором (при відсутності в ньому описів змінних), виконує основну роботу всередині цієї функції. Ті змінні, котрі описані в заголовку блоку (тобто в тілі функції) будуть локальними для цієї функції, тому що область їхньої видимості обмежена відповідним блоком. Вони створюються в момент звертання до даної функції і зникають по закінченні її роботи (клас пам'яті auto), а їхні імена не повинні збігатися з іменами формальних параметрів. Оскільки пам'ять під розміщення локальних змінних виділяється виконуючою системою динамічно з програмного стека, останній повинний мати достатню для цього довжину. Ініціалізація локальних об'єктів припустима лише у випадку простих змінних і неможлива для масивів і інших структурованих даних. Докладна розмова про це піде в Лекції 7. Виконання інструкцій у тілі функції починається з найпершого оператора і продовжується доти , доки не зустрінеться оператор повернення return, або поки не буде досягнутий кінець зовнішнього блоку. Значення, що повертається функцією, дорівнює значенню виразу в операторі return (див. Лекцію 3, $ 4), а при його відсутності вважається невизначеним. У разі потреби тип результату перетвориться до типу функції стандартним чином. Інструкція виклику функції в загальному випадку має наступний формат:

```
expression (<expression-list>),
```

і в сутності являє собою вираз, що може відігравати роль операнда в складі більш складного виразу. Тут expression є деякий адресний вираз, наприклад, ім'я функції, що визначає її вхідну точку. Список фактичних параметрів expression-list містить довільні вирази, розділені комами, значення яких обчислюються в момент звертання до функції і копіюються в область її формальних параметрів. Таким чином, у мові Сі реалізований механізм передачі параметрів за значенням. Оскільки усяка функція працює лише з копіями значень своїх аргументів, а не з їх адресами, то ніякі зміни значень формальних параметрів у тілі функції не можуть відбитися на значеннях фактичних параметрів. Це у свою чергу означає, що аргументи функції є носіями лише вхідної інформації і не можуть бути використані для передачі результатів її роботи функції, що викликала. Для подолання цього обмеження необхідно використовувати покажчики як аргументи функцій, передаючи тим самим числові значення відповідних адрес. У такий же спосіб вирішується проблема передачі масивів, функцій і деяких інших структурованих даних. Для ілюстрації викладеного матеріалу приведемо приклад програми, що обчислює квадратний корінь з числа, введеного з клавіатури консольного термінала. Тут основна робота з підрахунку кореня виконується функцією sqrt(), що реалізує ітераційний метод Ньютона з фіксованим числом ітерацій.

```
#include  <stdio.h>
main()
{ float  dat;
  float  sqrt(float);     /*  Опис функції (див. $ 3)  */
  printf("\nзадайте позитивне дійсне число ... ");
  scanf("%f", &dat);
  printf("\n\nкорень з числа %.3f дорівнює %.3f", dat, sqrt(dat));
}

float  sqrt(arg)
float  arg;
{ int    count;
  float  root = arg/2.0;
  for (count = 1; count <= 5; count++)
     root = 0.5*(root + arg/root);
  return (root);
}
```

Якщо тепер у відповідь на запит програми ввести, наприклад, число 25, то по закінченні її роботи буде надрукований наступний результат:

Корінь з числа 25.000 дорівнює 5.000

Зазначимо, що значення змінної dat у головній функції ні при яких обставинах не може бути змінене при виконанні ітераційного алгоритму в тілі функції sqrt(), оскільки вся робота тут ведеться з копією значення, переданого через параметр arg. Використаний в цьому прикладі попередній опис функції sqrt() у тілі функції main() буде розглянутий в $ 3 цієї лекції.

2. Використання покажчиків в якості аргументів функцій

Незважаючи на те, що прийняте в мові Сі рішення про передачу параметрів за значенням є досить природним і забезпечує повну незалежність окремих функцій, у ряді практично важливих випадків воно виявляється занадто обмежувальним. Дійсно, при такому рішенні аргументи будь-якої функції можуть служити лише носіями вхідної інформації, у той час як часто виявляється необхідним надати їм статус вхідно – вихідних. З іншого боку, передача за значенням структурованих даних, наприклад масивів, виявляється мало ефективною через великі втрати часу на копіювання елементів переданої структури і невиправданої перевитрати ресурсу пам'яті на створення такої копії. Саме тому в переважній більшості мов програмування механізм передачі агрегатів даних за значенням ніколи не реалізується. І, нарешті, відомий нам спосіб передачі параметрів принципово не дозволяє

використовувати одні функцій як аргументи інших функцій. Можливість подолання усіх вище відзначених труднощів відкривається в зв'язку з застосуванням могутнього апарату покажчиків, що є повноправними змінними, значення яких можуть передаватися з однієї функції в іншу. Така передача значень покажчиків, власне кажучи, реалізує механізм доступу до змінних, визначеним поза тілом функції, по адресному посиланню, оскільки самі покажчики є носіями адрес цих змінних. Одержавши ж необхідну адресу, функція, у свою чергу, може використовувати його для доступу до відповідного значення. Розгляд питання про використання покажчиків як аргументів функцій ми почнемо з найпростішого випадку передачі адрес скалярних змінних. Нехай, наприклад, необхідно мати функцію, що заміняє значення свого аргументу на його абсолютну величину. Щоб головна функція могла одержати результат такої заміни, вона повинна надати нашій функції адресу відповідної змінної, тобто покажчик на її розміщення в пам'яті ЕОМ. Рішення цієї задачі може мати такий вигляд:

```
void  abs(arg)
int  *arg;   /*  Формальний параметр  */
{ *arg = (*arg >= 0) ? (*arg) : -(*arg);
  return;
}
```

Тепер у випадку звертання

```
abs(&value);
```

адреса змінної value буде передана в тіло функції abs(), що замінить числове значення, розміщене за цією адресою , на його абсолютну величину. Іншим уже знайомим нам прикладом передачі адрес через апарат формальних/фактичних параметрів може служити використання стандартної функції scanf() для відформатованого введення значень із клавіатури консольного терміналу. Дійсно, звертання виду

```
scanf("%d", &alpha);
```

робить змінну alpha доступною в тілі цієї функції безпосередньо через її адресу, у результаті чого введене числове значення буде відоме й у місці виклику. Найбільш важливу роль при розробці програм мовою Сі грає можливість передачі між окремими функціями масивів змінних і, зокрема , символьних рядків. Тут знову виявляється корисним апарат покажчиків, тому що для забезпечення доступу до масиву в тілі усякої функції їй досить передати адресу його нульового елемента, причому носієм останнього є саме ім'я цього масиву. У наступному прикладі функція summa(), що виконує підсумовування елементів числового масиву, одержує від її головної функції main() адресу початку масиву vector і загальну кількість його елементів:

```
main()
{
  float  s;
  float  vector[100];    /*  Визначення локального масиву  */
  float  summa();        /*  Опис функції (див. $ 3)      */
  .....
  .....
  .....
  s = summa(vector, 100);
  .....
  .....
  .....
}

float  summa(mas, n)
float  mas[];          /*  Опис формального масиву    */
int    n;
{ int    i;
  float  sum = 0;
  for (i = 0; i < n; i++)
     sum += mas[i];
```

```
 return (sum);
}
```

Звертаємо увагу на те, що при описі формального масиву mas у заголовку функції summa() його розмірність явно не вказана. У цьому, однак, і немає необхідності, оскільки фактична робота буде виконуватися тут над масивом vector, адреса початку і довжина якого передаються при звертанні до цієї функції. Більше того, замість опису масиву невизначеної довжини досить мати еквівалентний йому опис,

```
float  *mas;
```

що визначає покажчик на початок оброблюваного масиву. Використаний в цьому прикладі попередній опис функції summa() у тілі функції main() буде розглянутий в $ 3 дійсної лекції. Прикладом використання параметрів для передачі символьних рядків можуть служити функції strcmp(), strcpy(), strlen() і інші, введені в Лекції 3. Аргументами кожної з них є масиви елементів типу char, що ідентифікуються своїми початковими адресами. У цьому випадку вже не потрібно додатково передавати кількість оброблюваних символів, тому що кінець усякого рядка легко знаходиться по закінчуючому його нуль-символу. Розглянемо нарешті можливість використання покажчиків для передачі одних функцій в інші, визначивши попереднє поняття покажчика на функцію. Останнє, мабуть, повинне зв'язуватися з адресою першого оператора, що виконується, відповідної функції, що задає її вхідну точку. Покажчик на функцію визначається в програмі подібно тому, як і самі функції, з тією лише різницею, що в цьому випадку заголовок виглядає в такий спосіб:

```
<sc-specifier> <type-specifier> (*identifier) (<parameter-list>)
<parameter-declarations>
```

Тут identifier суть ім'я обумовленого покажчика на функцію, а всі інші позначення використані в змісті своїх колишніх значень. Власне кажучи, приведений тільки що запис є окремим випадком розглянутої вище схеми, у якій declarator варто розуміти як вираз виду (*identifier). У наступному фрагменті програми

```
float  (*calc)(alpha, beta)
float  alpha, beta;
{ .....
  .....
  .....
}
```

ім'я calc визначене як покажчик на функцію, що повертає значення типу float. Звертання ж до цієї функції будуть виглядати так

```
(*calc)(x, y);
```

де змінні x і y мають тип float і відіграють роль фактичних параметрів. Покажчики на функції, на відміну від самих функцій, можуть не тільки виступати в ролі самостійних одиниць програми, але й бути формальними параметрами інших функцій. Саме ця їхня особливість дозволяє здійснити передачу вхідних адрес функцій через механізм параметрів. Для ілюстрації такої можливості розглянемо фрагмент програми, що обчислює проекцію відрізка довжини len, що складає кут alpha з віссю обсцис, на один із двох координатних напрямків у залежності від значення ключа direct:

```
double  len;                    /*  Довжина відрізка       */
double  alpha;                  /*  Кут з віссю x          */
main()
{ char    direct;               /*  Ключ напрямку          */
  double  px, py;               /*  Довжини проекцій       */
  double  cos(double), sin(double);   /*  Описи функцій,що  */
  double  proect(double (*) (double)); /*  викликаються (див. $ 3)   */
  ..............................
  ..............................
  switch (direct)
```

```
  { case 'x':
          px = proect(cos);       /*  Проекція на вісь x     */
          break;
    case 'y':
          py = proect(sin);       /*  Проекція на вісь y     */
          break;
  }
  .....................................
  .....................................
}

double  proect(func)
double  (*func)(double);            /*  Покажчик на функцію  */
{ return (len*(*func)(alpha)); }
```

У цьому прикладі формальним параметром функції proect є покажчик func на функцію, що повертає значення типу double. Фактичними ж параметрами при її виклику стають імена функцій cos() і sin(), що задають відповідні вхідні точки.

3. Попередній опис функцій

Через те, що окремі функції, що входять до складу будь-якої програми мовою Сі, є абсолютно незалежними одна від іншої в семантичному відношенні, кожна з них може бути визначена в довільному місці вихідного файлу, в окремому файлі чи знаходитися в одній із зовнішніх бібліотек. Така організація Сі-програми, що володіє великою гнучкістю і цілком відповідає вимогам структурного програмування, у визначеному змісті ускладнює здійснення контролю з боку компілятора за взаємною відповідністю типів і кількості формальних і фактичних параметрів, а також за правильністю типу значення, що повертається тій чи іншій функції. Причина цього полягає в тому, що будь-яка функція може бути викликана в роботу раніш, ніж вона визначена в поточному файлі, і в такій ситуації компілятор не має можливості стежити за правильністю виклику. Це твердження тим більше справедливе при звертанні до зовнішніх і бібліотечних функцій. У подібних випадках компілятор намагається інтерпретувати виклик деяким стандартним чином, привласнюючи значенню, що повертається функцією, тип int і не виконуючи перевірки правильності списку аргументів. У той же час, можливу невідповідність типів і кількості формальних і фактичних параметрів чи відмінність типу значення, що реально повертається функцією від типу int може призвести до помилок, що важко виявляються, у програмі. Простий вихід з цього положення полягає в складанні попереднього опису, чи прототипу функції, що викликається, у якому будуть визначені основні її атрибути.У самому загальному випадку такий попередній опис має наступний формат:

```
<sc-specifier> <type-specifier> declarator (<arg-list>)
                        <, declarator(<arg-list>) ... >;
```

Тут sc-specifier задає клас пам'яті (static чи extern), що має функція, яка викликається, (див. Лекцію 7, $ 2), type-specifier встановлює тип значення, що повертається їй, а arg-list визначає кількість і тип аргументів. Declarator у приведеній схемі є ідентифікатором функції, можливо модифікованим за допомогою круглих дужок і символу зірочки для покажчиків на функції і функцій, що повертають покажчики. Список аргументів arg-list звичайно являє собою послідовність розділених комами імен типів даних, підтримуваних наявним компілятором мови чи попередньо визначених за допомогою інструкції typedef (див. Лекцію 8, $ 8). Однак він також може містити і звичайні описи змінних виду float g. Використання ключового слова void на місці arglist дозволяє оголосити функцію, що не має параметрів. Оскільки функції в мові Сі можуть повертати будь-яке значення, крім масивів і інших функцій, typespecifier може задавати будь-який простий тип даних, а також структуру, чи об'єднання перерахування (див. Лекцію 8). У тих випадках, коли функція не виробляє ніякого значення чи повертає покажчик невизначеного типу (див. Лекцію 4, $ 1), її попередній опис повинний починатися з

ключового слова void, що знаходиться на місці імені типу значення, що повертається. Введене в такий спосіб поняття попереднього опису функції дає можливість компілятору побудувати деякий шаблон цієї функції до її фактичного визначення в поточному чи зовнішньому файлі. Цей шаблон може бути використаний для контролю правильності типу значення, що повертається функцією і відповідності формальних і фактичних параметрів. Нижче приведені кілька характерних прикладів побудови попередніх описів.

1. У цьому прикладі описана функція з ім'ям add, що повертає значення типу double і обидва аргументи якої є покажчиками на тип float:

```
double  add(float*, float*);
```

2. Якщо функція з ім'ям sum має два аргументи типу double і повертає покажчик на масив трьох елементів типу double, то її попередній опис повинний мати наступний вид:

```
double  (*sum(double, double))[3];
```

3. У тому випадку, коли функція не має аргументів і повертає покажчик невизначеного типу, у її попередньому описі необхідно використовувати ключове слово void на місці імені типу значення, що повертається, і списку аргументів:

```
void  *draw(void);
```

4. Аргументи командного рядка

Ті, кому хоч раз доводилося працювати в операційному середовищі MS DOS, звернули увагу на те, що більшість команд інтерфейсу, яким користуються програмісти можуть мати один чи більш параметрів, які називаються *аргументами командного рядка*. Так, наприклад, звертання до команди copy, що виконує копіювання файлів, звичайно виглядає таким чином:

```
copy  oldfile.txt  newfile.txt ,
```

де параметри oldfile.txt і newfile.txt визначають імена *файлу-джерела* і *файлу-приймача* відповідно. Ці параметри обробляються командним процесором і передаються в тіло програми copy, у результаті чого остання дізнається про файли, над якими повинна бути виконана операція копіювання. Можна з упевненістю сказати, що саме наявність параметрів робить усілякі команди зручними для практичного використання. Оскільки мова Сі часто застосовується при розробці системного програмного забезпечення, вона має вбудовані засоби для одержання аргументів команди безпосередньо від командного процесора. Така можливість, на перший погляд здається трохи незвичайною, у дійсності знаходиться в повній відповідності з архітектурою цієї мови. Справді , будь-яка функція, що входить до складу Сі-програми, може мати параметри, через які вона одержує необхідну інформацію від головної функції. Зовсім аналогічно, головна функція main(), з яким починається виконання всякої програми, могла б у момент виклику одержувати вхідні дані через аргументи командного рядка. Для цього досить постачити функцію main() набором параметрів, що звичайно мають імена argc і argv:

```
main(argc, argv)
```

Параметр argc (ARGument Count) є змінною типу int, що одержує від командного процесора інформацію про кількість аргументів, набраних у командному рядку, включаючи й ім'я самої команди. Другий параметр argv (ARGument Vector) звичайно визначається як масив покажчиків типу char, кожний з який зберігає адресу початку окремого слова командного рядка. Їхній опис у програмі може виглядати в такий спосіб:

```
int   argc;
char  *argv[];
```

Відповідно до прийнятої угоди, нульовий елемент argv[0] масиву покажчиків посилається на рядок символів, що містить ім'я самої команди і тому параметр argc завжди має значення більше чи рівне одиниці. Наступний елемент argv[1] задає адреса розміщення в пам'яті першого аргументу команди, також представленого послідовністю символів, і т.д. Звертаючись до чергового елемента масиву argv неважко одержати доступ до всіх аргументів командного рядка. Як приклад , що ілюструє

роботу з параметрами функції main(), розглянемо програму, що виводить на екран термінала свої власні аргументи і реалізує команду echo у складі оболонки операційної системи MS DOS:

```
#include <stdio.h>
main(argc, argv)
int  argc;
char  argv[];
{ int i;
  for (i = 1; i <= argc; i++)
    printf("%s%c", argv[i], (i < argc-1) ? ' ' : '\n');
}
```

Помістивши тепер завантажувальний модуль цієї програми у файл echo.exe і звернувшись до неї за допомогою команди

```
C:\> echo  first second third
```

одержимо на екрані термінала таке повідомлення

```
first second third
C:\>                    ,
```

що представляє собою просте відлуння аргументів командного рядка. Оскільки масив покажчиків у визначеному змісті еквівалентний "покажчику на покажчик" (див. Лекцію 4, $ 3), ми могли б визначити змінну argv у заголовку функції main() як непрямий покажчик типу char:

```
char  **argv;
```

що цілком рівносильно попередньому опису. У цих термінах наша програма echo могла б виглядати, наприклад, у такий спосіб:

```
#include <stdio.h>
main(argc, argv)
int  argc;
char  **argv;
{ while (-iargc > 0)
    printf((argc > 1) ? " %s " : "%s\n", *++argv);
}
```

де вираз ++argv збільшує на одиницю значення покажчика, змушуючи його посилатися на черговий рядок, отриманий від командного процесора.

Лекція 6

Введення і виведення у мові Сі: загальні концепції. Файли даних і каталоги файлів. Внутрішня організація і типи файлів. Стандартні функції для роботи з файлами і каталогами. Зовнішні пристрої як спеціальні файли. Організація обміну зі стандартними зовнішніми пристроями. Операції введення/виведення через порти мікропроцесорів Intel 8086/80286.

1. Введення і виведення у мові С: загальні концепції

При розробці й експлуатації програмного забезпечення операції введення/виведення грають настільки ж велику роль, як, наприклад, інструкції керування чи способи опису оперативних структур даних. Справа в тім, що, з одного боку, усяка програма, позбавлена можливості спілкування з "зовнішнім світом", виявляється або *недостатньо гнучкою* стосовно наборів даних, що нею оброблюються, або *абсолютно марною,* через відсутність можливості повідомити користувачу про результати своєї роботи. З іншого боку, у багатьох прикладних задачах обсяги інформації, що переробляється,

настільки великі, що не може бути і мови про її одночасне розміщення в оперативній пам'яті ЕОМ і виникає пряма необхідність у беззупинному "підкачуванні" даних у процесі роботи програми. І, нарешті, використання одних лише оперативних структур даних не забезпечує збереження інформації після вимикання живлення для ЕОМ, ліквідуючи тим самим можливість ведення довгострокових архівів і створення баз знань. У сучасних обчислювальних системах зовнішні пристрої, з якими приходиться підтримувати обмін, по виконуваних функціях поділяються на дві категорій. Це, *по-перше,* усілякі пристрої для зв'язку з оператором ЕОМ, як інтерактивні термінали і телетайпи, принтери, графобудівники, фотоскладальні пристрої і т.д. *По-друге*, зовнішніми стосовно центрального ЕОМ є накопичувачі інформації на магнітних носіях, якими найчастіше виступають магнітні диски. Ці дві категорії периферійних пристроїв принципово відрізняються тим, що в першому випадку передача даних на пристрій здійснюється послідовним чином (потоком) і носить, як правило, *односторонній характер*. В другому ж випадку звичайно має місце *двостороння передача інформації* і досить складна логічна організація даних на магнітному носії, підтримувана спеціальними компонентами операційної системи і прикладних програм. Однієї з основних особливостей мови Сі стосовно більшості інших мов програмування високого рівня є повна відсутність у ній визначених операторів введення/виведення. Це, однак, ні в якій мірі не обмежує його можливостей по організації великих наборів даних і керуванню периферійним устаткуванням, тому що всі необхідні для цього засоби надаються функціям зі стандартних бібліотек мови Сі. Більш того, подібне відношення до операцій введення/виведення відкриває шлях до створення високо ефективного і мобільного програмного забезпечення, що максимально використовує можливості операційної системи по підтримці зовнішнього обміну.

2. Файли даних і каталоги. внутрішня організація і типи файлів

Файлами прийнято називати пойменовані набори даних на зовнішніх носіях. Сукупність файлів звичайно організується по ієрархічному принципу у виді файлового дерева, чим забезпечується зручність представлення й обробки великих інформаційних структур. По характеру збереженої інформації розрізняють *файли даних* і *файли-каталоги (директорії).* Перші з них містять дані, безпосередньо використовувані прикладними програмами, включаючи вихідні тексти, об'єктні і завантажувальні модулі самих цих програм. Навпаки, вміст каталогів складають покажчики на інші файли і підкаталоги. Прикладні програми позбавлені можливості безпосередньо втручатися в структуру каталогів, а можуть робити це лише опосередковано шляхом звертання до відповідних функцій операційної системи. Той каталог, що знаходиться на вищому рівні ієрархічної організації, зветься *кореневим каталогом*. Логічна організація файлів даних у каталоги підтримується файловою системою, що входить до складу ОС. Основне призначення файлової системи полягає в тому, щоб відокремити користувача ЕОМ від зайвих деталей зовнішнього обміну і надати йому можливість простого і гнучкого керування великими інформаційними структурами. Іменами файлів і каталогів у Сі-програмах, призначених для наступної обробки в середовищі IBM C/2, можуть бути будь-які правильні імена в змісті операційної системи MS DOS. В окремих випадках забезпечується можливість використання спеціальних метасимволов для генерації послідовності імен. В операційному середовищі MS DOS не існує якої-небудь визначеної внутрішньої структури файлів. Навпаки, всякий файл являє собою просту послідовність байт, кожний з який може містити або правильний код символу ASCII, або бути довільною комбінацією двійкових розрядів. Файли першого типу звичайно називають *текстовими файлами* і вони можуть бути візуалізовані, наприклад, на екрані відеотермінала. Файли ж другого типу, називаються *двійковими файлами,* можуть містити усередині себе символи, що не друкуються, чи керуючі послідовності, й у загальному випадку не підлягають візуальному перегляду. Помітимо, однак, що з погляду внутрішньої обробки інформації в системі MS DOS, не існує принципової різниці між текстовими і двійковими файлами, і текстовий режим є основним при організації обміну інформацією. Проте, є визначені відмінності в інтерпретації вмісту файлів деякими стандартними функціями при їхньому використанні в текстовому і двійковому режимах. Так, наприклад, у текстовому режимі комбінація символів CR/LF (hex-коди 0D і 0A відповідно) перетворюється при введенні в єдиний символ LF. У процесі ж виведення виконується зворотнє перетворення. При роботі в двійковому режимі подібні перетворення не виконуються.

3. Стандартні функції для роботи з файлами і каталогами

У цьому параграфі наводяться найбільш вживані функції керування введенням/виведенням зі стандартної бібліотеки мови Сі. По своєму функціональному призначенню вони розбиті на *три* категорії: *введення/виведення потоком, операції обміну низького рівня* і *керування файлами і каталогами.* Організація обміну зі стандартними зовнішніми пристроями розглянута в наступному параграфі.

3.1. Введення/виведення потоком символів

Функції введення/виведення потоком розглядають файли й елементи даних як послідовності окремих символів. Вони надають можливість буферизації інформації при пересиланні, а також забезпечують перетворення данних відповідно до заданої специфікації формату. Всякий потік, перед звертанням до нього, повинний бути відкритий за допомогою функції fopen для введення, чи виведення виконання обох цих операцій у текстовому чи двійковому режимі. Відкриваючи потік, функція fopen повертає покажчик типу FILE, що використовується для посилання на нього у всіх операціях введення чи виведення або керування покажчиком поточної позиції. Після завершення роботи з потоком, його варто закрити, використовуючи функцію fclose. У протилежному випадку це зробить операційна система по закінченні роботи програми. Операції читання і запису для потоку виконуються, починаючи з поточної позиції покажчика, зміщуючи останній на відповідне число позицій. Використовуючи функцію fseek, покажчик можна встановити в довільну позицію файлу перед виконанням чергової операції, а функція rewind позиціонує покажчик на початок файлу. Попередні описи функцій введення/виведення потоком, а також визначення відповідних констант, типів і структур поміщені у файл stdio.h, який необхідно включити в програму за допомогою директиви препроцесора #include (див. Лекцію 7, $ 4 і приклади програм).

Ім'я функції і призначення: fopen - відкриває файл із заданим ім'ям для введення/виведення потоком
Формат і опис аргументів:

```
FILE *fopen(const char *pathname, const char *type)
const char *pathname;     /* Ім'я файлу, що відкривається, */
const char *type;         /* Тип доступу до файлу     */
```

Параметр type являє собою рядок символів і визначає характер доступу до файлу:

"r" - існуючий текстовий файл відкрити для читання

"w" - текстовий файл відкрити для запису (у випадку існування файлу його вміст руйнується)

"a" - текстовий файл відкрити для запису в кінець файлу (у випадку відсутності файл створюється знову)

"r+" - існуючий файл відкрити для читання і запису

"w+" - текстовий файл відкрити для читання і запису (у випадку існування файлу його вміст руйнується)

Додавання символу 'b' у кінець кожної з цих рядків дозволяє відкрити двійковий файл для виконання відповідних операцій. Функція повертає покажчик типу FILE при нормальному завершенні операції і NULL у випадку виникнення помилки.

Приклад використання:

```
#include <stdio.h>
main()
{ FILE *stream;
  if ((stream = fopen("data", "r")) == NULL)
     printf("Помилка при відкритті файлу");
}
```

Ім'я функції і призначення: fclose- закриває файл, попередньо відкритий для введення/виведення потоком

Формат і опис аргументів:

```
int  fclose(stream)
FILE  *stream;        /*  Покажчик на відкритий файл  */
```

Значення, що повертається, дорівнює нулю при нормальному завершенні операції і EOF у випадку виникнення помилки. Приклад використання:

```
#include  <stdio.h>
main()
{ FILE  *stream;
  stream = fopen("data", "w+b");
  .....
  .....
  fclose(stream);
}
```

Ім'я функції і призначення: feof – перевіряє виникнення ситуації "кінець файлу" при введенні/виведенні потоком

Формат і опис аргументів:

```
int  feof(stream)
FILE  *stream;        /*  Покажчик на відкритий файл  */
```

Значення, що повертається, відмінне від нуля при спробі читання за межами файлу і дорівнює нулю в протилежному випадку. Приклад використання:

```
#include  <stdio.h>
char  string[100];
FILE  *stream;
void  process(char*);
main()
{ while (!feof(stream))
    if (fscanf(stream, "%s", string))
      process(string);
}
```

Ім'я функції і призначення: fgetc - читає черговий символ з вхідного потоку і збільшує значення покажчика

Формат і опис аргументів:

```
int  fgetc(stream)
FILE  *stream;        /*  Покажчик на відкритий файл  */
```

Значення, що повертається, дорівнює коду прочитаного символу при нормальному завершенні операції і EOF у випадку виникнення чи помилки при досягненні кінця файлу.

Приклад використання:

```
#include  <stdio.h>
main()
{ char  buffer[81];
  int   i, ch;
  FILE  *stream;
  for (i = 0; i < 80 && (ch = fgetc(stream)) != EOF; i++)
     buffer[i] = ch;
  buffer[i] = '\0'
}
```

Ім'я функції і призначення: fgets - читає рядок символів із вхідного потоку до першого символу \n включно (але не більш заданої максимальної кількості), додаючи нуль-символ у кінець прочитаного рядка

Формат і опис аргументів:

```
char  *fgets(string, n, stream)
char  *string;        /*  Буфер для розміщення про-   */
                      /*  читанного рядка             */
int   n;              /*  Максимальна кількість       */
                      /*  символів у рядку            */
FILE  *stream;        /*  Покажчик на відкритий файл  */
```

Значення, що повертається, дорівнює покажчику на рядок string при нормальному завершенні операції і NULL у випадку виникнення чи помилки при досягненні кінця файлу.
Приклад використання:

```
#include  <stdio.h>
main()
{ char  line[100], *result;
  FILE  *stream;
  .....................................
  .....................................
  result = fgets(line, 100, stream);
  .....................................
  .....................................
}
```

Ім'я функції і призначення: getw - читає з вхідного потоку чергове ціле число, задане в двійковому форматі, і збільшує значення покажчика.
Формат і опис аргументів:

```
int  getw(stream)
FILE  *stream;        /*  Покажчик на відкритий файл  */
```

Значення, що повертається, дорівнює прочитанному цілому числу при нормальному завершенні операції і EOF у випадку виникнення чи помилки при досягненні кінця файлу.
Приклад використання:

```
#include  <stdio.h>
main()
{ int   ivalue;
  FILE  *stream;
  ......................
  ......................
  ivalue = getw(stream);
  ......................
  ......................
}
     ----------
```

Ім'я функції і призначення: fputc - записує символ у вихідний потік і збільшує значення покажчика
Формат і опис аргументів:

```
int  fputc(c, stream)
int   c;              /*  Виведений символ            */
FILE  *stream;        /*  Покажчик на відкритий файл  */
```

Значення, що повертається, дорівнює коду записаного символу при нормальному завершенні операції і EOF у випадку виникнення помилки.
Приклад використання:

```
#include  <stdio.h>
main()
{ char  buffer[81];
```

```
    int   i;
    FILE  *stream;
    .....................................
    .....................................
    for (i = 0; i < 81 && buffer[i] != '\0'; i++)
      fpufc(buffer[i], stream);
    .....................................
    .....................................
  }
```

Ім'я функції і призначення: fputs - копіює рядок символів у вихідний потік
Формат і опис аргументів:

```
int  fputs(string, stream)
const char  *string; /*  Покажчик на виведений рядок  */
FILE        *stream; /*  Покажчик на відкритий файл     */
```

Значення, що повертається, дорівнює коду останнього символу рядка при нормальному завершенні операції, нулю для порожнього рядка і EOF у випадку виникнення помилки.
Приклад використання:

```
#include  <stdio.h>
main()
{ FILE  *stream;
  .....................................
  .....................................
  fputs("This is a string", stream);
  .....................................
  .....................................
}
```

Ім'я функції і призначення: putw - записує у вихідний потік двійкове представлення цілого числа і збільшує значення покажчика.
Формат і опис аргументів:

```
int  putw(binint, stream)
int   binint;  /*  Двійкове представлення цілого числа  */
FILE  *stream; /*  Покажчик на відкритий файл           */
```

Значення, що повертається, дорівнює binint при нормальному завершенні операції і EOF у випадку виникнення помилки.
Приклад використання:

```
#include  <stdio.h>
main()
{ FILE  *stream;
  ...................
  ...................
  ...................
  putw(0347, stream);
  ...................
  ...................
  ...................
}
```

Ім'я функції і призначення: fscanf - читає дані з вхідного потоку і заносить їх у комірки пам'яті, обумовлені аргументами функції.
Формат і опис аргументів:

```
int  fscanf(stream, fstring <, argument ...>)
FILE       *stream;   /*  Покажчик на відкритий файл  */
const char  *fstring;   /*  Рядок керування форматом  */
```

Рядок string керування форматом визначає спосіб інтерпретації вхідних даних. Вона може містити наступні поля:

- пробіли, символи табуляції (\t) чи переходу на новий рядок (\n), що дозволяють читати без збереження будь-які комбінації відповідних символів із вхідного потоку;
- довільні символи, що друкуються, ASCII, крім знака відсотка (%), використання яких дозволяє читати без збереження відповідні їм символи з вхідного потоку;
- специфікації формату, що починаються зі знака відсотка (%) і які потребують перетворення прочитанних даних до заданого типу. Аргументи функції fscanf є покажчиками на комірки пам'яті, куди необхідно помістити прочитані значення, і повинні мати той же самий тип, що заданий відповідними специфікаціями в рядку керування форматом. Будь-яка специфікація формату в загальному випадку має наступний вид:

```
%<*><width><h|l|L>type        ,
```

де кутовими дужками (<>) позначені необов'язкові елементи конструкції. Окремі поля специфікації можуть бути символами чи числами, що задають конкретний спосіб перетворення даних при введенні. Символ перетворення відповідний полю type, визначає тип чергового елемента даних, що читається з вхідного потоку. Цим елементом може бути окремий символ ASCII, рядок символів чи число (ціле чи дійсне). Нижче наведена таблиця символів перетворення, що мають спеціальне значення в середовищі IBM C/2, а також зазначений тип, що відповідає кожному з них, аргументу.

Символ	Елемент вхідного потоку	Тип аргументу
d	десяткове ціле	покажчик на int
D	десяткове ціле	покажчик на long
o	восьмеричне ціле	покажчик на int
O	восьмеричне ціле	покажчик на long
x	шістнадцятиричне ціле	покажчик на int
X	шістнадцятиричне ціле	покажчик на long
i	десяткове, восьмеричне чи шістнадцятиричне ціле	покажчик на int
I	десяткове, восьмеричне чи шістнадцятиричне ціле	покажчик на long
u	десяткове ціле без знака	покажчик на unsigned int
U	десяткове ціле без знака	покажчик на unsigned long
e, f, g, E, G	дійсне число	покажчик на float
c	символ ASCII	покажчик на char
s	рядок символів	покажчик на масив елементів типу char
p	значення виду xxxx:yyy, де x і y є шістнадцятиричні цифри	покажчик на довгий покажчик (тип far) невизначеного типу

Символ зірочка (*), що можливо входить у специфікацію формату, змушує систему обміну проігнорувати черговий елемент даних заданого типу у вхідному потоці. Необов'язкове поле width являє собою позитивне десяткове число, що задає максимальну кількість символів, що читаються з їхнього вхідного потоку і інтерпретуються у відповідності зі специфікатором type. Додаткові специфікатори h, l і L можуть бути використані для модифікації типу аргументу, визначеного полем

type. Їхня дія поширюється лише на довжину відповідного аргументу (short чи long). Значення, що повертається функцією fscanf, дорівнює кількості прочитаних і перетворених відповідно до заданого формату полів вхідного потоку. При спробі читання за межами файлу функція повертає значення EOF.
Приклад використання:

```
#include  <stdio.h>
main()
{ char   sym, string[81];
  int    ival;
  long   lval;
  float  fval;
  FILE   *stream;
  .............................
  .............................
  stream = fopen("data", "r");
  fscanf(stream, "%c", &sym);
  fscanf(stream, "%s", string);
  fscanf(stream, "%d", &ival);
  fscanf(stream, "%ld", &lval);
  fscanf(stream, "%f", &fval);
  .............................
  .............................
}
        ----------
```

Ім'я функції і призначення: fprintf - записує відформатовані дані у вихідний потік
Формат і опис аргументів:

```
int  fprintf(stream, fstring <, argument ...>)
FILE       *stream;    /*  Покажчик на відкритий файл  */
const char  *fstring;   /*  Рядок керування форматом  */
```

Рядок fstring керування форматом визначає спосіб перетворення даних при виведенні. Він може містити в собі звичайні символи, що копіюються у вихідний потік, Esc-послідовності (див. Лекцію 2, $
і специфікації формату, що починаються зі знака відсотка (%). Аргументи функції fprintf є простими змінними чи покажчиками, що задають набір виведених значень. Всяка специфікація формату в загальному випадку має наступний вид:

```
%<flags><width><.precision><h|l|L>type
```

де кутовими дужками (<>) позначені необов'язкові елементи конструкції. Окремі полючи специфікації можуть бути чи символами числами, що задають конкретний спосіб перетворення даних при висновку. Символ перетворення, що відповідає полю type, визначає тип чергового елемента даних, записуваного у вихідний потік. Цим елементом може бути окремий символ ASCII, рядок символів чи число (ціле чи дійсне). Нижче приведена таблиця символів перетворення, що мають спеціальне значення в середовищі IBM C/2.

Символ	Аргумент	Формат виводу
d, i	ціле число	десяткове ціле зі знаком
u	ціле число	десяткове ціле без знака
o	ціле число	восьмеричне ціле без знака
x	ціле число	шістнадцятиричне ціле без знака, що використовує "abcdef"
X	ціле число	шістнадцятиричне ціле без знака, що використовує "ABCDEF"
f	дійсне число	дійсне число зі знаком з фіксованою крапкою

e, E	дійсне число	дійсне число зі знаком в експонентній формі
g, G	дійсне число	дійсне число зі знаком в експонентній чи формі з фіксованою крапкою в залежності від величини порядку
c	символ ASCII	окремий символ
s	покажчик типу char	рядок символів

Символи, що займають поле flags у рядку формату, дозволяють керувати розміщенням виведеної інформації в межах поля, заданого параметром width, а також визначають спосіб виведення чисел зі знаком. Так, знак мінус (-) у цьому випадку вимагає вирівнювання символів, що друкуються, по лівій границі поля width (за умовчанням виробляється вирівнювання по правій границі), а знак плюс (+) змушує виводити зі знаком (+ чи -) як позитивні, так і негативні числа. Додатковий прапор у виді знака номера (#) може використовуватися для висновку початкового нуля у восьмеричному форматі і пари 0x чи 0X у шістнадцятиричному форматі, а також для обов'язкового друку десяткової крапки при виведенні дійсних чисел навіть при нульовому значенні поля precision. Параметр width є цілим позитивним десятковим числом, що визначає мінімальну кількість позицій, які відводяться під відповідний елемент даних у вихідному потоці. Якщо кількість символів у виведеному значенні менше width, то здійснюється його розширення пробілами ліворуч чи праворуч у залежності від значення поля flags. Для заміни пробілів на нулі необхідно забезпечити параметр width префіксом нуль (0). При відсутності цього параметра в рядку формату він автоматично одержує деяке стандартне значення. Цілочисельний позитивний параметр precision визначає: мінімальну кількість цифр, що друкуються, у записі цілого числа (при використанні символів перетворення i, d, u, o, x чи X), кількість цифр праворуч від десяткової крапки при друці дійсних чисел (перетворення по форматах f, e, E, g чи G) чи максимальну кількість символів у рядку (при висновку по форматі s). У випадку відсутності, цей параметр заміняється деяким стандартним значенням, що залежить від типу виведеного елемента даних. Додаткові специфікатори h, l і L можуть бути використані для задавання довжини виведеного аргументу (short чи long). Значення, що повертається функцією fprintf, дорівнює кількості надрукованих символів.

Приклад використання:

```
#include  <stdio.h>
#define  MAX  100
main()
{ int    i;
  float  length[MAX], width[MAX];
  FILE   *printer;
  ..........................................................
  ..........................................................
  ..........................................................
  printer = fopen("prn", "w");
  fprintf(printer, "\t\t\t*** Результати розрахунку ***\n\n");
  for (i = 0; i < MAX; i++)
     fprintf(printer, "%3d  length = %6.3f,  width = %6.3f\n",
                                  length[i], width[i]);
  ..........................................................
  ..........................................................
  ..........................................................
}
```

Ім'я функції і призначення: fseek - переміщає покажчик поточної позиції файлу.

Формат і опис аргументів:

```
int  fseek(stream, offset, origin)
FILE  *stream;        /*  Покажчик на відкритий файл   */
long  offset;         /*  Зсув у байтах від origin  */
int   origin;         /*  Початкова позиція покажчика  */
```

Ця функція переміщає покажчик, зв'язаний з відкритим файлом, на offset байт вперед чи назад відносного заданого параметром origin початку відліку. Останній може бути однією з наступних символічних констант, визначених у файлі stdio.h:

SEEK_SET - початок файлу
SEEK_CUR - поточна позиція покажчика
SEEK_END - кінець файлу

Помітимо, що припустимим є позиціювання покажчика за межами кінця файлу, однак його переміщення за початок файлу приводить до помилки. Значення, що повертається, дорівнює нулю при нормальному завершенні операції і відмінне від нуля в протилежному випадку.
Приклад використання:

```
#include <stdio.h>
main()
{ int  result;
  FILE *stream;
  stream = fopen("data", "r");
  .....
  .....
  .....
  result = fseek(stream, 0L, SEEK_SET);
  .....
  .....
  .....
}
```

Ім'я функції і призначення: rewind - встановлює покажчик поточної позиції на початок файлу

Формат і опис аргументів:

```
void  rewind(stream)
FILE  *stream;        /*  Покажчик на відкритий файл  */
```

Ця функція не повертає ніякого значення в точку виклику.
Приклад використання:

```
#include <stdio.h>
main()
{ int  dat_1 = 1, dat_2 = 7, dat_3, dat_4;
  FILE *stream;
  stream = fopen("data", "w+");
    /*     Запис даних у файл      */
  fprintf(stream, "%d %d", dat_1, dat_2);
    /*  Встановлення покажчика на початок  */
  rewind(stream);
    /*       Зчитування даних       */
  fscanf(stream, "%d %d", &dat_3, &dat_4);
  .....
  .....
  .....
}
```

3.2. Операції введення/виведення низького рівня

Функції введення/виведення низького рівня здійснюють обмін з файлами чи периферійними пристроями шляхом прямого звертання до відповідних функцій операційної системи *(системним викликам).* Вони не надають можливості буферизації інформації при пересиланні і не забезпечують

перетворення даних із внутрішнього машинного представлення в текстовий формат. Перед звертанням до існуючого файлу його необхідно попередньо відкрити за допомогою функції open для введення чи виведення виконання обох цих операцій у текстовому чи двійковому режимі. Новий файл може бути створений шляхом використання функції creat. Обидві ці функції повертають у точку виклику цілочисельний параметр (file handle), який варто використовувати для посилання на відкритий файл у всіх наступних операціях введення/виведення низького рівня чи при керуванні покажчиком поточної позиції. Після завершення роботи з файлом, його потрібно закрити шляхом виклику функції close. У противному випадку це зробить операційна система по закінченні роботи програми. Операції читання і запису низького рівня виконуються функціями read і write, відповідно починаючи з поточної позиції покажчика. Використовуючи функцію lseek, покажчик можна встановити в довільну позицію файлу перед виконанням чергової операції введення/виведення. Перевірка виходу за кінець файлу здійснюється функцією eof. Попередні описи функцій введення/виведення низького рівня поміщені у файл io.h. Крім цього, файли fcntl.h, sys\types.h і sys\stat.h містять визначення символічних констант, використовуваних окремими функціями. Для нормальної роботи програми всі ці файли необхідно включити в її вихідний текст за допомогою директиви препроцесора #include (див. Лекцію 7, $ 4 і приклади програм). Більш повну інформацію про введення/виведення низького рівня можна знайти в керівництві Language Reference, що входить у стандартний комплект постачання компілятора IBM C/2.

3.3. Керування файлами і каталогами

При розробці окремих прикладних програм і тим більше системного програмного забезпечення можуть виявитися корисними надані операційною системою додаткові можливості керування структурою каталогів і зміни імен існуючих файлів. Відповідні операції реалізуються спеціальними функціями, що входять до складу бібліотеки мови Сі, що дозволяє уникнути безпосереднього звертання до системних функцій ОС. У цьому розділі приведені короткі зведення про деякі функції розглянутої групи. Їхні попередні описи поміщені в стандартні файли stdio.h і direct.h (конкретно див. приклади використання окремих функцій), що при необхідності можна включити у вихідний текст програми, використовуючи директиву препроцесора #include.

Ім'я функції і призначення: remove - видаляє файл із заданим ім'ям.
Формат і опис аргументів:

```
int  remove(pathname)
const char  *pathname; /*  Повне ім'я  файлу, що видаляється,  */
```

Значення, що повертається, дорівнює нулю при нормальному завершенні операції і -1 у випадку відсутності в каталозі файлу з заданим ім'ям чи забороні на його видалення (read-only file).
Приклад використання:

```
#include  <stdio.h>
main()
{ int  result;
  .....
  .....
  .....
  result = remove("c:\\myfile");
  if (result == -1)
     printf("Помилка при видаленні файлу");
  .....
  .....
  .....
}
```

Ім'я функції і призначення: rename - змінює ім'я існуючого файлу чи каталогу.
Формат і опис аргументів:

```
int  rename(oldname, newname)
```

```
const char  *oldname;          /*  Старе ім'я файлу  */
const char  *newname;          /*  Нове ім'я файлу   */
```

Ця функція, що перейменовує існуючі файли і каталоги, дозволяє також переміщати окремі файли з одного каталогу в інший шляхом зміни їхніх повних імен. У той же час, операції переміщення файлів між різними пристроями і цілими каталогами не є припустимими. Значення, що повертається, дорівнює нулю при нормальному завершенні операції і відмінне від нуля у випадку виникнення помилки (невірне завдання імені файлу oldname, існування файлу з ім'ям newname чи відсутність можливості створити файл із таким ім'ям, спроба переміщення окремих файлів з одного пристрою на інше і т.д.).
Приклад використання:

```
#include  <stdio.h>
main()
{ int  result;
  ..................................................
  ..................................................
  ..................................................
  result = rename("c:\\programs\myprog", "c:\\myprog");
  ..................................................
  ..................................................
  ..................................................
}
```

Ім'я функції і призначення: chdir - змінює поточний каталог.
Формат і опис аргументів:

```
int  chdir(pathname)
char  *pathname;      /*  Ім'я нового поточного каталогу  */
```

Значення, що повертається, дорівнює нулю при нормальній зміні поточного каталогу і -1 у випадку виникнення помилки.
Приклад використання:

```
#include  <direct.h>
main()
{ ..............
  ..............
  ..............
  chdir("..\\..");
  ..............
  ..............
  ..............
}
```

Ім'я функції і призначення: mkdir - створює новий каталог.
Формат і опис аргументів:

```
int  mkdir(pathname)
char  *pathname;        /*  Ім'я створюваного каталогу  */
```

Значення, що повертається, дорівнює нулю при нормальному завершенні операції і -1 у протилежному випадку.
Приклад використання:

```
#include  <direct.h>
main()
{ int  result;
  ..........................
  ..........................
  result = mkdir("b:\\newdir");
  ..........................
```

```
    ..........................
}
```

Ім'я функції і призначення: rmdir - видаляє існуючий каталог.
Формат і опис аргументів:

```
int  rmdir(pathname)
char  *pathname;        /*  Ім'я  каталогу, що видаляється,  */
```

Каталог, що видаляється цією функцією, повинний бути порожнім і не може бути поточним чи кореневим каталогом. Значення, що повертається, дорівнює нулю при нормальному завершенні операції і -1 у випадку виникнення помилки.

4. Зовнішні пристрої як спеціальні файли. Організація обміну зі стандартними зовнішніми пристроями

В операційному середовищі MS DOS різні зовнішні пристрої послідовного доступу (термінали, пристрої друку, графобудівники і т.д.) розглядаються як *спеціальні файли*. Ця обставина грає надзвичайно важливу роль з погляду гнучкого й ефективного керування цими пристроями і загальної технології організації введення/виведення. Дійсно, для того, щоб передати дані на який-небудь зовнішній пристрій, досить записати їх у деякий стандартний файл, використовуючи, наприклад, звичайні функції введення/виведення потоком символів. При цьому немає необхідності знати в деталях технічні характеристики цього пристрою, тому що усі вони відомі операційній системі, до складу якої входять відповідні програмні драйвери. Зв'язок імен спеціальних файлів з конкретними зовнішніми пристроями звичайно встановлюється на рівні базової системи введення/виведення (BIOS). Запускаючи в роботу будь-яку програму, операційна система MS DOS автоматично відкриває п'ять стандартних файлів, орієнтованих на виконання операцій введення/виведення потоком. Ці файли мають визначені імена і характерні режими доступу, приведені в наступній таблиці.

Ім'я файлу	Пристрій	Режим доступу
stdin	стандартний пристрій введення(звичайно клавіатура термінала)	зчитування
stdout	стандартний пристрій виведення (звичайно екран термінала)	запис
stderr	стандартний пристрій для повідомлень про помилки (звичайно екран термінала)	запис
stdaux	стандартний допоміжний пристрій (звичайний пристрій, підключений по послідовному інтерфейсі COM1)	зчитування /запис
stdprn	стандартний пристрій друку	запис

Визначені імена спеціальних файлів власне кажучи є покажчиками на структуру FILE і їх припустимо використовувати на місці параметра stream у всіх функціях введення/виведення потоком (див. $ 3 даної лекції). Так, наприклад, для друку повідомлення на екрані термінала досить скористатися функцією fprintf, задавши в якості її першого парметра ім'я stdout:

```
fprintf(stdout, "Цей стандартний пристрій висновку");
```

З іншого боку, введення символу, що надійшов від клавіатури термінала, можна виконати за допомогою функції fgetc, адресувавши її до стандартного пристрою введення stdin:

```
sym = fgetc(stdin);
```

Помітимо, однак, що не всі операції, визначені для звичайних файлів, мають сенс для стандартних приладів введення/виведення. Наприклад, наступнє звертання до функції fscanf

```
fscanf(stdout, "%d", &intval);
```

не є припустимим, тому що стандартний пристрій виведення не може бути використано як пристрій введення. Помилковим також буде звертання до функції fseek при любому покажчику на спеціальний файл. Для зручності роботи зі стандартними пристроями stdin і stdout до складу бібліотеки мови Сі додатково включені функції, спеціально орієнтовані на обмін з цими пристроями. Їхні імена багато в чому схожі з іменами розглянутих у $ 3 функцій введення/виведення потоком і приведені в наступній таблиці.

Ім'я функції	Призначення	Еквівалентне звертання
getch()	читає черговий символ від клавіатури термінала без відлуння на екрані.	нема
fgetchar() getchar()	читають черговий символ зі стандартного пристрою введення.	fgetc(stdin)
fputchar(s) putchar(s)	записують символ s на стандартний пристрій виведення	fputc(s, stdout)
gets(str)	читає рядок символів str зі стандартного пристрою введення.	fgets(str, 256, stdin)
puts(str)	копіює рядок символів str на стандартний пристрій виведення.	fputs(str, stdout)
scanf(...)	читає відформотовані дані зі стандартного пристрою введення.	fscanf(stdin, ...)
printf(...)	записує відформотовані дані на стандартний пристрій висновку.	fprintf(stdout, ...)

Кожна зі згаданих тут функцій семантично подібна відповідній для неї загальній функції і має ті ж самі аргументи, крім покажчика на структуру FILE. Їхні попередні описи також поміщені у файл stdio.h (крім опису функції getch(), що знаходиться у файлі conio.h).

5. Операції введення/виведення через порти мікропроцесорів intel

Усі периферійні пристрої і зовнішня пам'ять підключені до системної шини мікро-эвм через спеціальні інтерфейси. Кожен такий інтерфейс має набір вбудованих регістрів,які називають *портами введення/виведення*, через які центральний процесор і пам'ять взаємодіють з відповідними зовнішніми пристроями. Одні порти призначені для буферування даних і називаються *буферними портами* чи *портами даних.* Інші служать для збереження інформації про стан пристрою чи інтерфейсу і називаються *портами стану*. Треті називаються *портами керування* і використовуються для одержання команд від центрального процесора.
У деяких комп'ютерах адреси пам'яті і портів включені в єдиний адресний простір і всі команди зі звертанням до пам'яті можуть звертатися і до портів. У системах же побудованих на базі мікропроцесорів Intel 8086/80286 допускається організація двох адресних просторів: *пам'яті і введення/виведення*. У цьому випадку любий порт забезпечується індивідуальним номером, що однозначно ідентифікує його серед всієї безлічі вихідних портів. Стандартна бібліотека компілятора IBM C/2 надає користувачам дві функції з іменами inp() і outp(), що здійснюють прямий обмін інформацією з портами мікропроцесора. Можливість виконання операцій введення/виведення низького рівня необхідна при розробці програмних драйверів пристроїв, однак вона може виявитися корисною і при підготовці прикладних програм, що вимагають взаємодії з периферійним устаткуванням у реальному масштабі часу. Попередні описи функцій inp() і outp() поміщені у файл conio.h.

Ім'я функції і призначення: inp - читає один інформаційний байт через вхідний порт із заданим номером
Формат і опис аргументів:

```
int  inp(port)
unsigned  port;        /*  Номер опитуваного порту  */
```

Приклад використання:

```
#include  <conio.h>
main()
{ char     result;
  unsigned  port = 0x64;
  result = inp(port);
  printf("Уміст порту 64H дорівнює %0x\n", (int)result);
}
```

Ім'я функції і призначення: outp - записує один інформаційний байт у вихідний порт із заданим номером

Формат і опис аргументів:

```
int  outp(port, value)
unsigned  port;          /*  Номер вихідного порту  */
int      value;          /*  Виведене значення     */
```

Значення, що повертається функцією outp, дорівнює параметру value.

Приклад використання:

```
#include  <conio.h>
main()
{ unsigned  port = 0x64;
  outp(port, 0x03);
}
```

Лекція 7

Загальна структура програми мовою Сі. Час існування і видимість змінних. Блоки. Класи пам'яті. Автоматичні, зовнішні, статичні і реєстрові змінні. Рекурсивні функції. Реалізація рекурсивних алгоритмів. Препроцесор мови Сі: файли, що включаються, символічні імена і макровизначення. Моделі пам'яті, підтримувані компілятором IBM C/2.

1. Загальна структура програми мовою Сі. Час існування і видимість змінних. Блоки.

Як відзначалося наприкінці попередньої лекції, всяка програма мовою Сі являє собою сукупність однієї чи більш функцій, кожна з який є незалежний набір описів і операторів, вкладених між заголовком функції і її кінцем. Та функція, з якої починається виконання програми, називається *головною функцією*. Вона власне кажучи є вхідною точкою програми і повинна мати визначене ім'я main(). Всі інші функції, що можливо входять до складу програми і виконують конкретну частину роботи з реалізації алгоритму рішення задачі, відіграють підлеглу роль і запускаються в роботу шляхом їх прямого чи опосередкованого (через інші функції) виклику з головної функції. Робота всієї програми звичайно закінчується по досягненні кінця головної функції, однак її виконання може бути припинене шляхом передачі керування операційній системі з довільної точки програми. Будь-яка функція, у тому числі і головна, може мати один чи більше формальних параметрів, що використовуються для передачі в її тіло необхідних числових значень у момент виклику. Наявність параметрів у головної функції є специфічною особливістю мови Сі і використовується для обробки

аргументів командного рядка при звертанні до програми. Незважаючи на те, що функції в мові Сі являють собою основні будівельні блоки всякої програми, при розгляді прикладів і задач попередніх лекцій нам уже приходилося зустрічатися з визначенням змінних, констант і літерних ланцюжків поза якої б то не було функції, чи, інакше кажучи, на *зовнішньому рівні.* Саме так ми поводилися при роботі з файлами, що включаються, які зберігали визначення системних констант і попередні описи функцій, чи бажаючи зробити яку-небудь змінну доступною відразу декільком функціям. Визначені на зовнішньому рівні об'єкти уже втрачають властивість локальності стосовно функцій і формують програмну оболонку, яка називається *глобальним середовищем*. Для подальшого вивчення структури Сі-програми, нам необхідно визначити поняття часу існування і видимості різних об'єктів, що беруть участь у її роботі. Під *часом існування змінної чи функції* звичайно розуміють тривалість їхнього реального перебування в пам'яті ЕОМ. Ті об'єкти програми, що створюються в момент початку її роботи й існують аж до повного завершення останньої, прийнято називати *об'єктами з глобальним часом існування*. З іншого боку, змінні і структури даних, що створюються динамічно в процесі роботи програми і ліквідовані раніше за її фактичне завершення, будемо називати *об'єктами з локальним часом існування*. Пам'ять для представлення локальних об'єктів виділяється компілятором у момент входу в той програмний компонент, що містить описи цих об'єктів, і повертається компілятором назад операційній системі по досягненні її кінця. Під *видимістю змінної чи функції* варто розуміти ту область вихідної програми, у межах якої дозволена робота зі змінною чи звертанням до відповідної функції. Об'єкти програми, звертання до яких дозволено з будь-якої її точки, називають *об'єктами з глобальною видимістю*. Такими, наприклад, є всі функції, що входять до складу програми, а також змінні й структури даних, визначені поза певною функцією. Навпаки, ті змінні, область доступу до яких обмежена деякою частиною вихідної програми, ми будемо називати *змінними з локальною видимістю*. Так, локальну видимість мають будь-які об'єкти, описані в тілі функції, якщо тільки не прийнято спеціальних мір для розширення області доступу до них. Загальне правило, що визначає час існування і видимість об'єктів програми, ґрунтується на фундаментальному понятті блоку. Під *блоком* у мові Сі розуміється сукупність описів і операторів, вкладених у фігурні дужки. Це поняття є прямим узагальненням введеного в Лекції 2 поняття складеного оператора. Принципова їхня відмінність полягає в тому, що до складу блоку порівняно з операторами, що виконуються, входять опис змінних чи структур даних. Відповідно до семантичних правил мови, любий елемент даних, визначений всередині блоку, має локальну видимість і час існування в рамках цього блоку. Іншими словами, елемент даних створюється в момент входу в блок і ліквідується при виході з нього, при чому доступ до цього елемента обмежений рамками поточного блоку. Тіло всякої функції в змісті нової термінології являє собою правильний блок із всіма обмеженнями, що звідси випливають, на час існування і видимість внутрішніх змінних даної функції. Про описи об'єктів програми в якому-небудь блоці ми будемо говорити як про *визначення цих об'єктів на внутрішньому рівні*. Навпаки, описи констант, змінних, структур чи даних функцій поза всяким блоком будемо називати *зовнішніми описами чи описами на зовнішньому рівні.* Ми вже говорили про те, що опис змінних у Сі-програмах дозволяється як на зовнішньому, так і на внутрішньому рівнях, причому час існування і видимість внутрішніх змінних обмежена поточним блоком. У випадку ж зовнішнього опису, змінна має глобальний час існування, а її видимість обмежена частиною програми від тієї точки, де зустрівся цей опис, до кінця поточного файлу. На відміну від описів змінних, визначення функцій може розміщатися лише на зовнішньому рівні, і тому всяка функція має глобальний час існування і глобальну видимість стосовно програми в цілому. Іншими словами, ніяка функція не може містити всередині себе визначення іншої функції з локальною видимістю. У той же час, всередині будь-якого блоку припустимий попередній опис функцій, використовуваних у цьому блоці, що, однак, не звужує області видимості таких функцій. Правильні блоки в програмі мовою Сі можуть бути вкладені один в інший, породжуючи тим самим *нелокальне середовище посилань*. При цьому внутрішній блок може містити описи змінних з тими ж іменами, що і в зовнішньому блоці. Це веде до *локального перевизначення* відповідних змінних у внутрішньому блоці. Колишні значення цих змінних відновлюються після передачі керування якому-небудь оператору зовнішнього блоку. У наступному фрагменті програми, що виконаєм підсумовування елементів двох цілочисельних масивів, ідентифікатор sum у внутрішньому і зовнішньому блоках позначає різні об'єкти, кожний з який має своє власне розміщення в пам'яті ЕОМ:

```
summa(tab_1, m, tab_2, n)
int  *tab_1, m;   /*  Адреса і розмірність першого масиву  */
```

```
int  *tab_2, n;    /*  Адреса і розмірність другого масиву  */
{ int  i, sum;
  sum = 0;
  for (i = 0; i < m; i++)
    sum = sum + tab_1[i];
      { int  sum;
        for (i = 0; i < n; i++)
          sum = sum + tab_2[i];
        printf("\nсумма елементів другого масиву = %d", sum);
      }
  printf("\nсумма елементів першого масиву = %d", sum);
}
```

Аналогічна ситуація виникає й у тому випадку, коли яка-небудь змінна, визначена всередині блоку, має те ж саме ім'я, що і деяка зовнішня змінна. Тут, так само як і раніше, локальний опис заміщає зовнішній опис змінної, відновлюючи її колишнє значення по завершенні роботи блоку. Вихідний текст Сі-програми допускається підрозділяти на кілька файлів, кожний з який містить необхідні описи змінних і визначення функцій. Компіляція всіх таких файлів повинна вироблятися роздільно з наступною зборкою відповідних об'єктних модулів на етапі побудови готової до виконання програми (див. Лекцію 1). Нижче під час обговорення питання про класи пам'яті, будуть сформульовані умови, при яких дозволені посилання на об'єкти за межами поточного файлу. Як приклад програми, що містить як зовнішні, так і внутрішні описи, приведемо реалізацію алгоритму сортування числового масиву по зростанню елементів методом перестановки:

```
#include  <stdio.h>   /*  Включення стандартного файлу stdio.h   */
#define  MAX  100    /*  Максимальна припустима кількість     */
                  /*  елементів у масиві                  */

float  mas[MAX];     /*  Зовнішній опис числового масиву     */

main()
{ int   i;          /*  Параметр циклів                     */
  int   n;          /*  Реальна кількість елементів масиву  */
  FILE  *fp;        /*  Покажчик на файл, що містить        */
                  /*  вхідний масив                       */

/*--------     Введення елементів масиву з файлу mas.dat     --------*/

  if ((fp = fopen("mas.dat", "r")) == NULL) return;
  n = 0;
  while (fscanf(fp, "%f", &mas[n]) != EOF)
    n++;
  fclose(fp);

/*---------            Сортування масиву             --------*/

  { int    mind;       /*  Індекс максимального елемента       */
    float  buf;        /*  Буфер для перестановки елементів    */
    for (i = n-1; i > 0; i--)
      { mind = maxind(i);
        buf = mas[i]; mas[i] = mas[mind]; mas[mind] = buf; }
  }                 /*  Кінець внутрішнього блоку           */

/*--------          Друк результатів              ----------*/

  printf("\t\t\t*** Масив впорядкований по зростанню ***\n\n");
```

```
  for (i = 0; i < n; i++)
    printf("%7.3f", mas[i]);
}                   /*  Кінець головної функції              */

/*--------   Перебування індексу максимального елемента    ----------*/

maxind(m)
int  m;
{ int    i;          /*  Параметр циклу                    */
  int    mind;       /*  Поточне значення номера макс. эл-та  */
  float  max;        /*  Поточне значення макс. елемента     */
  max = mas[0]; mind = 0;
  for (i = 1; i <= m; i++)
    if (mas[i] > max)
      { max = mas[i]; mind = i; }
  return (mind);
}                   /*  Кінець функції maxind               */
```

У цьому прикладі масив дійсних чисел mas визначений на зовнішньому рівні і тому доступний обом функціям програми. Змінні mind і buf визначені локально у внутрішньому блоці функції main() і область їхньої видимості обмежена одним цим блоком. Те ж саме ім'я mind використане в тілі функції maxind() для позначення поточного значення індексу максимального елемента і ніяким чином не пов'язане з його визначенням функції main(). Аналогічне зауваження можна зробити і щодо параметра циклу i у головній, і допоміжній функціях.

2. Класи пам'яті

Одна з переваг мови Сі полягає в тому, що вона дозволяє керувати ключовими механізмами програми. Поняття класів пам'яті представляє приклад такого керування. Вони дають можливість визначити, з якими функціями зв'язані певні змінні і як довго той чи інший об'єкт програми зберігається в пам'яті ЕОМ. Іншими словами, призначаючи клас пам'яті для якого-небудь елемента, програміст визначає тим самим час існування цього елемента (глобальне чи локальне). Область же видимості елемента даних характеризується рівнем (зовнішнім чи внутрішнім), на якому він визначений у програмі. Незважаючи на те, що в мові Сі існують лише два принципово різних механізми представлення об'єктів програми - глобальний і локальний, прийнято, проте , виділяти чотири основних класи пам'яті, використовуючи для їхнього призначення в програмі наступні ключові слова:

auto extern static register

які називаються *описувачами класу пам'яті*. Останні звичайно з'являються в інструкціях опису даних перед ім'ям типу, модифікуючи семантику відповідного опису. Вони впливають не тільки на час існування, але і на область видимості того чи іншого об'єкта програми. Значення описувачів класу пам'яті будуть різними в залежності від рівня програми (зовнішнього чи внутрішнього), на якому вони зустрічаються, а також від того, чи застосовані вони при описі змінної чи функції. Нижче розглянуті чотири можливих випадки призначення класу пам'яті для різних об'єктів програми.

2.1. Автоматичні змінні (клас пам'яті auto)

Об'єкти програми, що належать класу пам'яті auto називаються зазвичай *автоматичними змінними*, мають локальний час існування й область їхньої видимості обмежена тим блоком, у якому вони визначені. Пам'ять для їхнього представлення виділяється динамічно з програмного стека в момент входу в цей блок і повертається назад системі по досягненні його кінця. За умовчанням всі змінні,

описані на внутрішньому рівні (тобто в тілі функції), є автоматичними. Однак це можна явно підкреслити, використовуючи ключове слово auto:

```
main()
{ auto int   alpha, beta;
  auto char  line[81];
  .....
  .....
  .....
}
```

Оскільки будь-які змінні, описані на зовнішньому рівні, повинні мати глобальний час існування і розміщаються в статичній пам'яті, використання описувача auto поза яким-небудь блоком є неприпустимим і призводить до помилки. По цій же причині не дозволяється призначати клас пам'яті auto для функцій, тому що їхні визначення не можуть бути поміщені в тіло програмного блоку. Ініціалізація простих автоматичних змінних задається звичайним чином (див. Лекцію 3, $ 3) і виконується при кожнім вході у відповідний блок. У випадку ж відсутності ініціалізуючого виразу в інструкції опису даних, значення будь-якої такої змінної вважається невизначеним.
Ініціалізація будь-яких агрегатованих даних (наприклад, масивів), що мають клас пам'яті auto, не підтримується компілятором мови Сі.
Зауваження. Оскільки пам'ять для представлення автоматичних змінних виділяється з програмного стека, при побудові готової до виконання програми з об'єктних модулів необхідно задати розмір стека таким, щоб його вистачило для розміщення всіх цих змінних у найбільшому по обсязі даних програмному компоненту.

2.2. Зовнішні об'єкти програми (клас пам'яті extern)

Зовнішніми об'єктами програми прийнято називати змінні, визначені поза якою-небудь функцією, і самі ці функції, оскільки визначення однієї функції в тілі іншої не є припустимим. За умовчанням всяка зовнішня змінна має глобальний час існування, а область її видимості лежить між точкою опису в програмі і кінцем поточного файлу. Це означає, що будь-яка функція, визначена після опису деякої зовнішньої змінної в межах одного файлу, може використовувати цю змінну без певних додаткових оголошень. Якщо ж виникає необхідність у використанні зовнішніх змінних до їхнього фактичного визначення в поточному чи навіть іншому файлі, їх потрібно явно описати, тобто, що вони мають клас пам'яті extern у тілі відповідної функції. У наступному прикладі:

```
main()
{ extern int  number;    /*  Опис зовнішньої змінної     */
  .....
  .....
  .....
}
.....
.....
.....
int  number = 17;        /*  Визначення зовнішньої змінної  */
.....
.....
.....
```

змінна number використовується функцією main() раніш, ніж вона визначена в програмі. Її початкове значення в цій функції дорівнює 17. З погляду розглянутих понять часу існування і видимості об'єктів програми, важливо розрізняти описи зовнішніх змінних і їхніх визначень. Так, описи починаються з ключового слова extern і можуть зустрічатися багаторазово на зовнішньому і внутрішньому рівнях, задаючи тим самим область видимості відповідних змінних. Навпаки,

відсутність описувача extern при визначенні зовнішньої змінної жадає від системи фактичного виділення пам'яті для її розміщення. Будь-яка така змінна може бути визначена лише один раз у всій сукупності файлів, що утворять вихідну програму. Ініціалізація зовнішніх змінних і структур даних завжди припустима в інструкціях, що визначають ці об'єкти в програмі, і виконується один раз перед початком її роботи. Ті елементи даних, для яких ініціалізуючий вираз явно не заданий, за умовчанням ініціалізується нулем. Інструкції опису зовнішніх змінних, що починаються з ключового слова extern, не можуть містити ініціалізуючих виразів. Використання зовнішній змінних у програмах мовою Сі часто виявляється корисним у тих випадках, коли виникає необхідність у передачі великих обсягів інформації між окремими функціями чи коли безліч функцій використовують ті самі дані. В обох цих випадках використання апарата формальних/фактичних параметрів приведе до зайвої громіздкості і поганої структурованості програми. Однак невиправдане розширення області видимості великої кількості змінних шляхом їхнього усуспільнення може знизити надійність програмного забезпечення.

2.3. Статичної змінні і функції (клас пам'яті static)

Описувач класу пам'яті static може використовуватися в інструкціях визначення даних як на зовнішньому, так і на внутрішньому рівнях. Зовнішні статичні змінні мають глобальний час існування, однак на відміну від звичайний зовнішній змінних, доступ до них обмежений частиною поточного файлу, що лежить після визначення будь-якої такої змінної в програмі. Посилання на статичні змінні з іншого файлу чи їхнє використання до фактичного визначення в поточному файлі не припустимі. Таким чином, описувач static дозволяє звузити область видимості зовнішньої змінної. Внутрішні статичні змінні, як і змінні класу auto, мають локальну видимість у межах поточної функції (чи блоку), однак вони зберігаються в пам'яті ЕОМ протягом усього часу роботи програми. Це дозволяє створювати об'єкты з глобальним часом існування, обмежуючи їхню видимість у програмі. Так, наприклад, змінна count, визначена в тілі функції calc:

```
main()
{ .....
  .....
  .....
}

calc()
{ static int  count;
  .....
  .....
  .....
}
```

може бути використана тільки в межах цієї функції, хоча вона і зберігає своє значення після завершення роботи останньої. Клас пам'яті static можна також призначати і при визначенні функцій, обмежуючи тим самим можливість доступу до них межами одного файлу. Це дозволяє мати в різних файлах функції з однаковими іменами, не побоюючись виникнення конфліктної ситуації. Ініціалізація статичних змінних і структур даних припустима як на зовнішньому, так і на внутрішньому рівнях, і виконується по тим же правилам, що і для звичайних зовнішніх змінних.

2.4. Реєстрові змінні (клас пам'яті register)

Реєстрові змінні з погляду видимості, часу існування і можливостей ініціалізації цілком рівносильні автоматичним змінним. Однак вони зберігаються не в оперативній пам'яті машини, а на робочих регістрах центрального процесора, що підвищує ефективність виконання програми. Якщо можливість такого збереження відсутня (наприклад, через зайнятість цих регістрів), вони стають звичайними

автоматичними змінними. Цей клас пам'яті може призначатися тільки лише для внутрішніх змінних і формальних параметрів функцій за допомогою описателя register. Наприклад:

```
func(p, q)
register int  p, q;
{ register char  sym;
  ...................
  ...................
  ...................
}
```

Операція одержання адреси для реєстрових змінних не припустима. На багатьох типах ЕОМ цей клас пам'яті не може бути призначений змінним типу float.

3. Рекурсивні виклики функцій. Реалізація рекурсивних алгоритмів

Будь-яка функція в мові Сі має реєнтерабельний (повторний вхідний) код, що дозволяє їй звертатися до самої себе чи безпосередньо через інші функції. Такі звертання називаються *рекурсивними чи викликами рекурсії* При кожному черговому рекурсивному виклику створюється нова копія параметрів функції, а також визначених у її тілі автоматичних і реєстрових змінних. Зовнішні і статичні змінні, що мають глобальний час існування, зберігають при цьому свої колишні значення і розміщення пам'яті. Незважаючи на те, що ні стандарт мови Сі, ні компілятор IBM C/2 формально не накладають ніякого обмеження на кількість рекурсивних звертань, проте воно практично завжди існує для будь-яких типів ЕОМ, тому що кожен новий виклик вимагає додаткової пам'яті з ресурсу програмного стека. Якщо кількість викликів занадто велика, виникає переповнення сегмента стека й операційна система вже не може створити чергового екземпляра локальних об'єктів функції, що веде, як правило, до аварійного завершення програми. Як приклад реалізації рекурсивного алгоритму розглянемо функцію printd(), що друкує ціле число у виді послідовності символів ASCII (тобто цифр, що утворять запис цього числа):

```
printd(num)
int  num;              /*   число, що  друкується,        */
{ int  i;
  if (num < 0)
     { putchar('-'); num = -num; }
  if ((i = num/10) != 0)
     printd(i);          /*  Рекурсивний виклик функції  */
  putchar(num % 10 + '0')
}
```

Якщо значення змінної value дорівнює 123, то у випадку виклику

```
printd(value);
```

ця функція двічі звернеться сама до себе для друку цифр заданого числа.

4. Препроцесор мови Сі

Препроцесор мови Сі призначений для внесення змін у вихідний текст програми безпосередньо перед її компіляцією, а також для встановлення значень параметрів, що використовуються компілятором. Звертання до препроцесора здійснюється за допомогою набору директив, що починаються із символу номера (#), і дозволяє домогтися добірності і мобільності Сі-програм. У цьому параграфі розглядаються деякі найбільш важливі і часто використовувані директиви препроцесора, що працює в складі компілятора IBM C/2.

4.1. Директива #define

Ця директива, що дозволяє привласнювати текстовим рядкам символічні імена, може мати один з наступних форматів:

```
#define identifier <text>
```

чи

```
#define identifier(parameter-list) <text>  ,
```

де кутовими дужками (<>) позначена необов'язкова частина конструкцій. Перша форма інтерпретується препроцесором як проста літерна підстановка, виконувана перед фактичним звертанням до компілятора. Друга ж з них служить для створення *макровизначень*, семантично подібних до визначення звичайних функцій. Тут identifier є довільне, правильне в змісті граматики мови, ім'я (див. Лекцію 1, $ 3), а text являє собою рядок символів, що заміщає всяке входження цього імені у вихідний текст програми. Помітимо, що така заміна виконується лише для окремих імен і не має місця в тих випадках, коли identifier входить до складу текстових рядків чи є складовою частиною більш довгого ідентифікатора. Список аргументів parameter-list містить у собі один чи більше формальних параметрів, розділених комами й унікальними іменами. У цьому випадку препроцесор заміняє імена параметрів, що входять у text, іменами відповідних фактичних аргументів, і лише після цього буде виконана реальна підстановка тексту на місце ідентифікатора. Символьний рядок text може мати довільну довжину, що можливо навіть перевершує довжину одного рядка екрана. Для формування довгого тексту необхідно набрати символ \, натиснути клавішу Enter і продовжити введення символів з початку нового екранного рядка. З іншого боку, параметр text може взагалі бути відсутнім у складі директиви, що відповідає визначенню ідентифікатора identifier без присвоєння йому якого-небудь конкретного значення. Приведемо кілька прикладів використання директиви #define для завдання імен текстових ланцюжків і формування макровизначень функцій.

1. Найбільш характерним застосуванням цієї директиви є призначення символічних імен констант:

```
#define  WIDTH   100
#def1ne  LENGTH  (WIDTH + 30)
```

Тут ідентифікатор WIDTH оголошений числовою константою, рівної 100, а ім'я LENGTH визначене за допомогою WIDTH і цілого числа 30, причому круглі дужки в другому випадку є істотними.

2. Програмісти, що бажають бачити свою програму синтаксично допоможе, наприклад, на вихідний текст програми мовою Алгол-60, можуть скористатися директивою #define для заміни ключових слів і спеціальних символів, прийнятих у мові Сі:

```
#define  begin    {
#define  end      ;}
#define  integer  int
#define  real     float
#define  then
```

У цьому прикладі остання інструкція визначає ідентифікатор then, не привласнюючи йому ніякого значення. Ефектом такого визначення є простий витяг з тексту програми усіх входжень імені then, що дозволяє записати умовний оператор у такому виді

```
if (a >= b) then
begin
  max = a;
  min = b;
end
```

не порушуючи тим самим синтаксичних правил мови Сі.

3. Замість визначення функції abs(x), що обчислює абсолютну величину змінної x, у програмі можна мати семантично еквівалентне йому макровизначення виду

```
#define  abs(x)    ((x) >= 0) ? (x) : (-x)
```

звертання до якого ззовні нічим не буде відрізнятися від звертання до звичайної функції. Директива #define може зустрічатися в довільному місці файлу, що містить вихідний текст Сі-програми, однак звичайно їх прийнято поміщати в початок цього файлу.

4.2. Директива #undef

Дана директива має формат

```
#undef identifier
```

і змушує препроцесор ігнорувати всі наступні входження визначеного раніше в інструкції #define імені identifier. Це дозволяє обмежити область вихідної програми, у межах якої identifier обробляється препроцесором і має спеціальне значення. У наступному прикладі область дії ідентифікаторів WIDTH і ADD обмежена лише частиною вихідного файлу, у межах якої вони приймаються в увагу препроцесором мови Сі:

```
#define  WIDTH     100
#define  ADD(x, y)  (x) + (y)
.....
.....
.....
#undef  WIDTH
#undef  ADD
.....
.....
```

Інструкція #undef може бути поміщена в довільне місце файлу, що містить вихідний текст програми.

4.3. Директива #include

Формат цієї директиви в загальному випадку визначається наступною схемою:

```
#include  "pathname"
```

чи

```
#include  <pathname>
```

де кутові дужки (<>) є складовою частиною конструкції. Тут pathname є правильне ім'я файлу в змісті операційної системи MS DOS. Директива #include дозволяє включати текст файлу з ім'ям pathname до складу файлу, що містить звертання до цієї директиви. Якщо pathname задає повне ім'я файлу від кореневого каталогу, то дві приведені вище форми запису директиви еквівалентні. У тім же випадку, коли задане відносне ім'я файлу, використання подвійних лапок (" ") змушує препроцесор насамперед шукати необхідний файл у поточному каталозі, потім переглядати каталог, визначений у команді виклику компілятора і, нарешті, у разі потреби продовжити пошук у стандартному каталозі. Однак при виведенні імені файлу в кутові дужки поточний каталог не проглядається. Препроцесор допускає послідовні вкладення інструкцій #include максимально до десяти рівнів. Це означає, що усякий файл, що підключається за допомогою цієї директиви, може містити усередині себе нове звертання до #include. У складі вихідного тексту програми директива #include може розміщатися в довільному місці утримуючого її файлу.

5. Моделі пам'яті, підтримувані компілятором ibm c/2

Сегментна організація доступу до оперативної пам'яті ЕОМ, характерна для мікропроцесорів Intel і тісно зв'язана з особливостями їхньої внутрішньої архітектури, вимагає спеціального розгляду питання про розміщення програмного коду і даних у процесі роботи програми. Це питання має важливе значення з погляду формування виконавчих адрес при звертанні до пам'яті, і його вивчення необхідне для повного розуміння апарата роботи з покажчиками і механізму передачі параметрів між окремими функціями. Всяка програма мовою Сі після її підготовки до виконання (тобто компіляції і редагування зв'язків) і наступного завантаження в пам'ять персонального комп'ютера IBM PC представляється трьома основними типами незалежних сегментів: *коду, даних і стека*. Розмір одного сегмента кожного з цих типів не може перевищувати 64K байт пам'яті (1K = 1024 байт). Принципово для нормальної роботи будь-якої програми необхідно мати принаймні два таких сегменти, один із яких зберігає інструкції виконуваної програми (*сегмент коду*), а другий містить дані (*сегмент даних*), оброблювані цією програмою. Однак якщо програма досить велика чи великим є набір оброблюваних нею даних, то може знадобитися більш одного сегмента того чи іншого типу. Крім цього, у більшості випадків виявляється необхідним мати спеціальним чином організований сегмент стека для збереження автоматичної змінної і передачі параметрів функціям при їхньому виклику. Власне кажучи обговорювані нижче моделі організації пам'яті мають пряме відношення до кількості сегментів різного типу і визначають тим самим максимально припустимий об'єм програми і її даних. Компілятор IBM C/2 підтримує п'ять основних моделей сегментації пам'яті: *small, medium, compact, large і huge*. Кожна з них може бути обрана за допомогою відповідної опції при виклику компілятора і забезпечується своїми власними бібліотеками стандартних функцій. Модель small звичайно використовується для невеликих по обсязі програм, машинний код яких і дані можуть розміститися в одному сегменті кожний. Загальний розмір програми в цьому випадку не повинний перевищувати 128K байт пам'яті. Така модель є найбільш економічною в змісті використання оперативної пам'яті і швидкості роботи програми. Для неї компілятор робить короткі 16-розрядні покажчики (тип near) при звертанні до даних і інструкцій програми. Модель medium дозволяє мати великий обсяг програмного коду, але допускає роботу лише з відносно невеликою кількістю даних. У цьому випадку код програми може займати довільна кількість сегментів по 64K кожен, однак сегмент даних повинний бути лише один. Окремі сегменти коду створюються компілятором з незалежних програмних модулів, якими є файли, що містять вихідний текст програми. Для такої моделі інструкції програми адресуються за допомогою 32-розрядних покажчиків (тип far), а дані вибираються в межах єдиного сегмента за допомогою покажчиків типу near. Модель compact у визначеному змісті протилежна попередній моделі, тому що для неї характерний великий обсяг даних, що займають довільну кількість стандартних сегментів, і малий обсяг програмного коду, що поміщається компілятором в один сегмент. У розглянутому випадку доступ до даних здійснюється за допомогою покажчиків типу far, а для вибору з пам'яті інструкцій програми досить 16-розрядної адреси. Модель large допускає створення великих програм, що складаються з довільної кількості сегментів коду і даних, загальний обсяг яких обмежений лише адресним простором використовуваної ЕОМ (1Мбайт для мікропроцесора Intel 8086 і 16Мбайт для мікропроцесора Intel 80286) і реальним об'ємом оперативної пам'яті. Однак незважаючи на це, створення масивів змінних, перевищуючих 64Кбайт, не є припустимим для розглянутої моделі. У даному випадку всі покажчики мають тип far. Модель huge цілком ідентична моделі large з тією лише різницею, що вона дозволяє готувати масиви змінних, потребуючи для свого збереження більш 64Кбайт пам'яті, розміщаючи кожний з них у декількох сегментах. Раціональне використання п'яти розглянутих моделей пам'яті дає можливість створювати гнучкі, ефективні і великі по обсязі програмні системи. Більш того, припустиме довільне змішання цих моделей шляхом ігнорування стандартних угод про довжину адрес і явного призначення типу покажчиків за допомогою ключових слів near, far і huge при їхньому визначенні в програмі. Так, наприклад, декларація

```
int  (far *vector)[30];
```

визначає покажчик типу far на масив vector тридцяти елементів типу int незалежно від того, яка модель пам'яті буде обрана при компіляції програми.

Лекція 8

Структури в мові Сі: основні поняття. Масиви структур. Покажчики на структури. Вкладення структур. Структури і функції. Об'єднання. Перерахування. Визначення і використання нових типів даних. Класи імен.

1. Структури в мові Сі: основні поняття

При рішенні задач обчислювальної математики, інформаційного забезпечення і системного програмування дуже часто приходиться зіштовхуватися з наборами даних, що мають досить складну логічну організацію. Такими, наприклад, є всілякі каталоги, статистичні таблиці, інформаційні бюлетені і набори даних, створювані компілятором при обробці програм на мовах високого рівня. У той же час, оперативна пам'ять ЕОМ організована вкрай примітивно в логічному відношенні, представляючи собою послідовність занумерованих осередків довжиною в один байт кожна. Така невідповідність логічної складності виникаючих у додатках структур даних і можливостей їхнього машинного представлення неминуче повинне викликати значні труднощі при розробці програмного забезпечення, змушуючи програмістів всякий раз вирішувати непросту задачу відображення цих структур даних на лінійну структуру пам'яті ЕОМ. Можливий шлях рішення виникаючої проблеми складається в створенні мов програмування, що підтримують таку логічну організацію даних, що найбільш типова для широкого кола прикладних задач. До подібних мов, зокрема , відноситься і мова Сі, що володіє надзвичайно могутніми засобами представлення й обробки складних агрегатів даних. У Лекціях 3 і 4 ми вже розглянули одно- і багатомірні масиви змінних, визначивши їхній як упорядковані послідовності елементів даних одного типу. Не викликає ніяких сумнівів, що поняття масиву є одним з найважливіших понять всякої мовної системи, призначеної для формалізованого опису обчислювальних алгоритмів. Однак статичність і однорідність масивів у тім виді, як вони були визначені раніше, трохи обмежує можливість їхнього застосування для опису внутрішніх логічних зв'язків реальних інформаційних систем. У цій лекції розглядаються більш складні агрегати даних, названі структурами, для яких вимога однорідності вже не має місця. Самі ж структури можуть бути об'єднані в масиви чи вкладатися одна в іншу, надаючи тим самим можливість надзвичайно гнучкої логічної організації даних. Під *структурою* в мові Сі розуміється набір однієї чи більшого числа змінних, що можливо мають різний тип і об'єднаних під єдиним ім'ям, названим *ім'ям структури*. Останнє повинно бути правильним ідентифікатором у змісті граматики мови (див. Лекцію 1, $ 3). Опис всякої структури в програмі починається з ключового слова struct і в найпростішому випадку має наступний формат:

```
struct { member-declaration list } identifier <, identifier ... >;
```

де struct є ключове слово мови Сі, а в кутові дужки (<>) укладена необов'язкова частина конструкції. Тут member-declaration list є один опис, чи більше описів змінних, кожна з який називається *елементом структури*, а identifier - суть ім'я змінної, обумовленої як така, що має тип *структура*. Так, наприклад, інструкція

```
struct { char name[30];
         int group; } student;
```

визначає структуру з ім'ям student, елементами якої є масив символів name і цілочисельна змінна group. Кожне з описів у member-declaration list має той же самий формат, що і розглянуті раніше описи звичайних змінних чи масивів, однак тут неприпустиме використання описувачів класу пам'яті (див. Лекцію 7, $ 2) та ініціалізуючих виразів (див. Лекцію 3, $ 3). Уся ця сукупність описів, вкладена у фігурні дужки, визначає загальну схему структури і називається її *шаблоном.* Всякий елемент структури, що входить до складу шаблона, повинний мати своє власне, унікальне в межах даного шаблона, ім'я. Пам'ять під розміщення окремих елементів структури виділяється компілятором послідовно, починаючи з першого з них, чим гарантується безупинне збереження структури в цілому. Так у нашому прикладі, цілочисельна змінна group, що займає два байти, буде розміщена відразу ж після останнього елемента символьного масиву name. Доступ до елементів структури здійснюється шляхом вказівки її ідентифікатора, за яким випливає символ крапка (.), і імені конкретного елемента цієї структури. Така операція зветься *операція одержання елемента структури*, а її пріоритет

так само високий, як і пріоритет операції доступу до окремих елементів масиву (див. Лекцію 3, $ 1). Наприклад, позначення

```
student.group
```

задає елемент group у складі структури student, у той час як посилання

```
student.name[9]
```

виділяє десятий елемент масиву name тієї ж структури. На відміну від імені масиву, ім'я структури саме по собі не є синонімом свого імені. Власне кажучи це означає, що вживання імені структури без наступного за ним символу операції одержання елемента (.) і його імені не є припустимим і приводить до помилки на етапі компіляції програми. Для одержання ж адреси початку структури необхідно явно застосувати операцію & (див. Лекцію 4, $ 1) до імені чи структури її першого елемента. Наприклад, наступні два адресних вирази

```
&student    i    &student.name[0]
```

цілком еквівалентні і їхні значення дорівнюють адресі розміщення структури student у пам'яті ЕОМ. Зовсім аналогічно, вираження

```
&student.group
```

визначає адресу елемента group у складі розглянутої структури. Розглянутий нами найпростіший спосіб визначення структур вимагає всякий раз повторювати шаблон структури в кожнім новому описі, навіть якщо ці описи визначають структури з однієї і тією же загальною схемою. Щоб уникнути такого повторення, мова Сі надає можливість включання в шаблон створюваної структури деякого імені, що називається *тегом структури*. Тому більш загальний спосіб опису структур у програмі має наступний формат:

```
struct <tag> { member-declaration list } <identifier <, ... >>;
```

де tag є ім'ям, що присвоюється шаблону структури, а всі інші позначення зберігають свій колишній зміст. Після свого визначення в програмі, *тег структури* разом із ключовим словом struct може бути використаний як синтаксичний еквівалент імені типу даних, тобто припустимими є наступні описи виду

```
struct tag identifier <, identifier ... >;
```

що визначають ім'я identifier як таке, що має тип struct tag. У наступному прикладі

```
struct DATE  { int   day;
               int   month;
               int   year;
               char  day_name[15];
               char  mon_name[15]; }  date_1;
```

ім'я DATE привласнюється шаблону структури й одночасно визначається конкретна структура date_1, під представлення якої компілятором виділяються тридцять шість байт оперативної пам'яті. Після цього для визначення структури date_2 вже немає необхідності знову описувати шаблон структури, а досить скористатися скороченою формою виду

```
struct DATE  date_2;
```

Частковим випадком розглянутої тільки що загальної схеми визначення структур є опис, у якому відсутній список імен змінних після закриваючої фігурної дужки, але в обов'язковому порядку зазначене ім'я в поле tag. Наприклад:

```
struct STACK  { int    pointer;
                float  vector[100]; };
```

Такий опис визначає лише загальну схему структури, привласнюючи їй ім'я STACK і не вимагаючи від компілятора фактичного виділення пам'яті. Тепер, скориставшись скороченою формою, можна визначити конкретну структуру

```
struct STACK  stack;
```

що має раніше зафіксований шаблон. Посилання ж на окремі елементи структури stack задаються в цьому випадку звичайним чином

```
stack.vector[9]
```

що відповідає звертанню до десятого елементу масиву vector у складі структури stack. В інструкціях опису структур, як і при визначенні звичайних змінних чи масивів, може міститися додаткова інформація про клас пам'яті (див. Лекцію 7, $ 2) у виді відповідного описувача, що коштує перед ключовим словом struct:

```
static struct STACK  memory;
```

Способи їхнього призначення і роль у програмі залишаються в цьому випадку колишніми. Зокрема, структури, що мають клас пам'яті static чи extern, можуть бути ініціалізовані шляхом вказівки у фігурних дужках списку початкових значень усіх чи декількох перших елементів обумовленої структури. Так, наприклад, повертаючи до структурного шаблона з ім'ям DATE, можемо записати

```
struct DATE  date_3 = { 17, 3, 1989,
                        "п'ятниця",
                        "лютий" };
```

У тому випадку, коли список ініціалізованих виразів менше повної кількості елементів у структурі чи відсутній взагалі, елементи зовнішніх і статичних структур, що не мають ініціалізатора, покладаються рівними нулю. При описі ж структур, що мають клас пам'яті auto, значення їхніх елементів залишаються невизначеними.

2. Масиви структур

Окремі структури з довільним загальним шаблоном, як і звичайні змінні будь-якого типу, можуть бути об'єднані в масиви фіксованої довжини. З погляду загальної теорії баз даних така операція відповідає завданню обмеженого вектора, елементами якого є об'єкти деякого абстрактного структурованого типу. Описи масивів структур у програмі будуються на тій же самій синтаксичній основі, що й описи звичайних масивів. Так, у наступному прикладі

```
struct BOOK  { char  author[30];  /*  Автор книги        */
               char  title[256];  /*  Назва книги       */
               int   year;        /*  Рік видання       */
               int  pages;        /*  Кількість сторінок  */
             }  catalog[10];      /*  Масив структур    */
```

ім'я catalog об'явлено як масив десяти структур із загальним шаблоном BOOK. Організація даних, подібна цієї, може бути використана, наприклад, при складанні бібліографічних каталогів. Для звертання до окремих елементів масиву структур його ім'я всякий раз необхідно модифікувати за допомогою квадратних дужок ([]), що задають *операцію узяття елемента масиву*. Конкретний елемент обраної з масиву структури виділяється в цьому випадку звичайним чином за допомогою символу крапка (.). Так, звертання виду

```
catalog[3].title[4]
```

задає п'ятий символ масиву title елементів типу char у складі четвертого елемента масиву структур catalog. Застосовуючи до якого-небудь елемента масиву операцію & одержання адреси

```
&catalog[7]
```

можна знайти початок розміщення в пам'яті відповідної структури.

Зауваження. Оскільки ім'я всякого масиву є синонімом своєї адреси, то згадування самого по собі імені масиву структур тотожно операції одержання адреси нульового елемента цього масиву. Зокрема , для масиву структур catalog з попереднього приклада справедливі рівності:

```
catalog == &catalog[0].author[0] == catalog[0].author
```

3. Покажчики на структури

У попередньому параграфі, визначаючи поняття масиву структур, ми зробили перший крок до розуміння того, що тип struct є зовсім повноправним типом даних мови Сі. Тепер, розширюючи наші представлення про структури, спробуємо визначити поняття покажчика на структуру і додати йому конкретний зміст у програмі. Формально покажчик на структуру можна описати подібно тому, як ми це робили для покажчиків на прості типи даних. Загальна схема такого опису повинна мати один з наступних форматів:

```
struct <tag> { member-declaration list } *identifier <, ... >;
```

чи

```
struct tag *identifier <, ... >;
```

що знаходиться в повній відповідності із синтаксичними правилами складання описів у мові Сі. Зокрема, повертаючись до прикладів з попередніх параграфів, ми могли б написати

```
struct STUDENT  { char  name[30];
                  int   group; }  *studptr;
```

визначаючи тим самим покажчик studptr на структурний тип STUDENT. При цьому комбінація *studptr повинна розглядатися як сама структура і формально можна задати посилання на її окремий елемент, використовуючи операцію крапка (.):

```
(*studptr).group
```

Помітимо, що в цьому прикладі, як і при визначенні покажчиків на масиви (див. Лекцію 4, $ 3), круглі дужки є істотними, оскільки стандартний пріоритет *операції одержання елемента (.)* вище пріоритету *операції непрямої адресації (*)*. Однак останній запис може і не мати конкретного змісту, оскільки створюючи покажчик на структуру компілятор не виділяє реальну пам'ять під збереження її елементів. Ця ситуація цілком аналогічна тієї, з яким ми зіштовхнулися в Лекції 4, розглядаючи еквівалентність масивів і покажчиків. У той же час, покажчики на структури забезпечують принципову можливість більш гнучкого маніпулювання даними, ніж самі структури, оскільки використовуючи апарат покажчиків пам'ять під розміщення елементів структури можна виділяти динамічно за допомогою функцій *alloca(), malloc() чи realloc()* (див. Лекцію 4, $ 4). Так, наприклад, скориставшись першої з цих функцій, ми можемо написати

```
studptr = (struct STUDENT*)alloca(n*sizeof(struct STUDENT));
```

зарезервувавши тим самим блок пам'яті, достатній для розміщення масиву n структур із шаблоном STUDENT. Тепер, використовуючи позначення останнього приклада, спробуємо додати конкретний зміст арифметичним операціям над покажчиками на структури. Згадуючи, що збільшуючи на одиницю значення покажчика на простий тип даних, ми змушували його посилатися на черговий елемент даних відповідного типу, неважко зрозуміти, що операції виду

```
studptr = studptr + 1    чи    studptr++
```

зміщають покажчик на структурний тип STUDENT на початок чергової структури цього типу. Використовуючи далі загальне поняття еквівалентності масивів і покажчиків, можна проіндексувати покажчик на структуру, розуміючи цю операцію в змісті рівності адрес

```
studptr + i == &studptr[i]
```

чи в контексті виділення окремого елемента структури

```
(*(studptr+i)).name[k] == studptr[i].name[k]
```

Для зручності виділення елементів структур, заданих своїми покажчиками, у мові Сі додатково введена *операція проходження*, знаком який є комбінація -> символів '-' і '>'. Її пріоритет збігається з

пріоритетом звичайної операції одержання елемента структури. Використовуючи цю операцію в нашому прикладі, замість позначення

```
(*studptr).name
```

що визначає адресу масиву name у складі структури з покажчиком studptr, варто писати

```
studptr->name
```

що семантично зовсім еквівалентно попередньому запису.

4. Вкладення структур

Продовжуючи далі розгляд питання про використання структур для побудови складних логічних агрегатів даних і згадуючи зауваження попереднього параграфа про повноправність типу struct, варто допустити можливість включення одних структур як елементів шаблона інших. Така схема визначення структури власне кажучи є ні що інше, як *вкладення структур* і її досить повно ілюструє наступний приклад:

```
struct  { int     pointer;
          struct  { int    length;
                    float  array[MAX}; }  vector[DEPTH];
        }  stack;
```

в якому масив структур vector є внутрішнім стосовно структури stack. З погляду загальної теорії баз даних приведений опис може визначати стек динамічних обмежених векторів елементів типу float. У цьому випадку елемент pointer структури stack варто розглядати як покажчик вершини стека глибини DEPTH, реалізованого на базі масиву структур vector. Кожна ж структура цього масиву визначає динамічний вектор, що довжина його в цей момент дорівнює length і обмежена зверху константою MAX. Для посилання на окремі елементи вкладених структур необхідно керуватися загальним правилом доступу до полів структури,
послідовно застосовуючи операцію одержання елемента. Так, наприклад, оператор виду

```
stack.vector[stack.pointer].length++;
```

збільшує на одиницю поточну довжину динамічного вектора, розташованого на вершині стека. Трохи більш складним прикладом вкладення структур є їхнє рекурсивне визначення, побудоване таким чином, що *як шаблон внутрішньої структури використовується шаблон зовнішньої структури.* Подібні агрегати даних можуть застосовуватися, наприклад, для представлення *бінарних дерев, алгоритми пошуку* для який носять яскраво виражений рекурсивний характер. Дійсно, структура з наступним шаблоном

```
struct TNODE  { char         word[30];
                struct TNODE  *left;
                struct TNODE  *right; };
```

може визначати вузол бінарного дерева, у якому розташовується символьний масив word і з який можна спуститися вниз по дереву в двох можливих напрямках left і right.

5. Структури і функції

Зовсім очевидно, що окремі елементи структур, що є простими змінними чи покажчиками довільного типу, можуть бути використані як аргументи при звертанні до функцій. Однак більш важливим є питання про можливість передачі через апарат формальних/фактичних параметрів структур у цілому. Цю операцію найбільше природно здійснити, використовуючи поняття покажчика на структуру. Для ілюстрації технічних деталей, зв'язаних з передачею й обробкою структур, розглянемо фрагмент програми, що відшукує в зведеному каталозі книгу, що має найбільш ранній рік видання. Загальна

організація даних, необхідна для рішення цієї задачі, може бути представлена за допомогою структурного шаблона BOOK, описаного в $ 2 дійсній лекції.

```
#include <stdio.h>
#define MAX 300
struct BOOK { char author[30]; /* Автор книги */
              char title[256]; /* Назва книги */
              int year;       /* Рік видання */
              int pages;      /* Кількість сторінок */
            };
main()
{ int    min_year;
  struct BOOK catalog[MAX];
  .....
  .....
  .....
  min_year = find(catalog);
  printf("\nнайстаріша книга видана в %d року", min_year);
}

int find(book)
struct BOOK *book;      /* Покажчик на структуру */
{ int cnt;              /* Параметр циклу */
  int min;              /* Поточне значення мінімуму */
  min = book->year;
  for (cnt = 0; cnt < MAX; cnt++, book++)
     if (book->year < min)
        min = book->year;
  return (min);
}
```

У цьому прикладі формальний параметр book функції find визначений як покажчик на структуру із шаблоном BOOK. При звертанні до даної функції їй передається адреса нульового елемента масиву структур catalog. Далі при кожній черговій ітерації циклу в тілі функції find() значення покажчика book збільшується на одиницю, після чого він посилається на наступну структуру в масиві catalog.
Зауваження. Деякі реалізації мови Сі, зокрема компілятор IBM C/2, допускають використання структур як єдиного цілого як аргументів функцій, передаючи за значенням окремі елементи таких структур.

6. Об'єднання

Об'єднання - це засіб, що дозволяє розміщати дані різних типів в одному й тому ж місці оперативної пам'яті. З погляду граматики мови Сі, всяке об'єднання є змінною, що приймає в різний час виконання програми значення різних типів. Описи об'єднань мають той же самий формат, що й описи структур і в загальному випадку задаються однією з наступних схем:

```
union <tag> { member-declaration list } <declarator <, ... >>;
```

чи

```
union tag declarator <, declarator ... >;
```

Тут union є ключове слово мови Сі, а інші позначення використані в змісті своїх колишніх значень. Declarator найчастіше є ідентифікатором змінної, можливо модифікованим за допомогою квадратних дужок (для масивів об'єднань) чи символу зірочка (для покажчиків на об'єднання). Пам'ять, що виділяється під об'єднання, визначається довжиною найбільшого елемента в складі даного об'єднання.

При цьому всі члени об'єднання зберігаються в одній і тій же області пам'яті з незмінною початковою адресою. Для ілюстрації використання об'єднань розглянемо приклад організації даних, що забезпечують доступ до робочих регістрів мікропроцесорів Intel 8086/80286.

```
union REGS  { struct  { unsigned int   ax;
                        unsigned int   bx;
                        unsigned int   cx;
                        unsigned int   dx;    }  x;
              struct  { unsigned char  al, ah;
                        unsigned char  bl, bh;
                        unsigned char  cl, ch;
                        unsigned char  dl, dh; }  h;
            }  regs;
```

Тут елементи внутрішніх структур x і h розміщені в одній і тій же області пам'яті, в яку буде пересилатися вміст робочих регістрів (скажемо, за допомогою програми, написаної мовою Ассемблера). При цьому звертання виду

```
regs.x.ax
```

моделює доступ до регістра AX мікропроцесора, а вирази

```
regs.h.al   і   regs.h.ah
```

забезпечують відповідно доступ до молодшого AL і старшому AH байтам цього регістра.

7. Перерахування

Тип даних перерахування дозволяє визначити набір символічних імен, зв'язавши з кожним з них числове значення цілого типу. Як і при визначенні структур, припустимими є дві загальні форми складання описів:

```
enum <tag> { enum-list } <identifier <, identifier ... >>;
```

чи

```
enum tag identifier <, identifier ... >;
```

Тут enum є ключове слово мови Сі, а під enum-list розуміється список розділених комами ідентифікаторів і необов'язкових виразів константного типу:

```
identifier <= cjnstant-expression>
```

Кожен ідентифікатор у цьому списку іменує один елемент із безлічі значень, що перелічуються. За умовчанням перший з них одержує значення нуль, другий - значення одиниця і т.д. Опція

```
= constant-expression
```

вхідна в enum-list, дозволяє порушити стандартне правило призначення числових значень іменам з цього списку, при чому константне вираження constant-expression повинне мати тип int. Припустимим також є збіг числових значень різних імен у списку. Теги перерахувань визначаються по тим же самим правилам, що і теги структур і об'єднань, і можуть бути використані в скороченій формі опису, заміщаючи enum-list. У наступному прикладі визначається перерахування з ім'ям day і з'являється змінна workday, що має тип перерахування:

```
enum day  { saturday,
            sunday = 0,
            monday,
            tuesday,
            wednesday,
            thursday,
```

```
friday } workday;
```

Значення нуль зв'язується з ідентифікатором saturday за умовчаванням, а ідентифікатору sunday воно ж привласнюється явно . Інші п'ять імен у цьому списку послідовно одержують значення від 1 до 5.

8. Визначення і використання нових типів даних

З можливістю визначення нових типів даних ми власне кажучи вже зустрічалися при описі структур, об'єднань і перерахувань. Дійсно, визначаючи, наприклад, тег структури, програміст фактично вводить у роботу і постачає деяким ім'ям нову організацію даних, причому посилання на неї стають припустимими всюди надалі як і на стандартні імена типів даних. На додаток до вже розглянутих засобів, у мові Сі мається спеціальна інструкція, що дозволяє розширити можливості визначення і використання нових типів даних, зробивши їх у той же час більш лаконічними. Ця інструкція починається з ключового слова typedef і в загальному випадку має наступний формат:

```
typedef type-specifier declarator <, declarator ... >;
```

Тут type-specifier є ім'я простого типу даних чи опис шаблона структури. Припустимо також є використання імен типів, визначених раніше в інструкції typedef. Declarator, як і колись, являє собою, можливо модифіковане, ім'я змінної, котре стає ім'ям знову обумовленого цією інструкцією типу даних. Розглянемо трохи найбільш характерних прикладів використання інструкції typedef.

1. У найпростішому випадку ця інструкція дозволяє перевизначити імена стандартних типів даних, призначивши, наприклад, ім'я whole для позначення стандартного типу int:

```
typedef int whole;
```

2. Найбільш характерним застосуванням інструкції typedef є призначення імен знову обумовленим агрегатам даних. Наступний опис

```
typedef struct { int poiter;
                 char string[81]; } STACK;
```

вводить новий тип даних з ім'ям STACK, що задає структуру із шаблоном із двох елементів.

У процесі розробки програм імена типів даних, визначені за допомогою інструкції typedef, можуть використовуватися всюди в рівній мірі з іменами стандартних типів даних.

9. Класи імен

У програмах мовою Сі ідентифікатори використовуються для іменування великого числа різних об'єктів, таких як константи, змінні, функції, теги й елементи структур, мітки і т.д.. З метою полегшення вибору імен і зниження ймовірності виникнення конфліктних ситуацій, компілятор мови підрозділяє всю безліч ідентифікаторів на взаємно *непересічні класи.* У межах того самого класу всяке ім'я повинне бути унікальним з урахуванням області видимості і часу існування зв'язаного з цим ім'ям об'єкта програми. З іншого боку, однакові імена можуть позначати безліч об'єктів, що належать різним класам, не приводячи при цьому до протиріччя на етапах компіляції, зборки і виконання програми. Нижче приводиться опис класів пам'яті, підтримуваних компілятором IBM C/2.

1. Найбільш великий і важливий клас складають імена змінних, функцій, формальних параметрів, що перелічуються констант і типів даних, що визначаються за допомогою інструкції typedef. Ідентифікатори, що позначають відповідні об'єкти, не повинні збігатися між собою на тому самому рівні програми.

2. Інший істотний клас утворять теги структур, об'єднань і перерахувань, імена яких повинні бути різними в межах одного рівня видимості. Однак допускається збіг імен тегів з іменами змінних, функцій, елементів структур і інших об'єктів із суміжних класів.

3. В окремий клас виділяються імена елементів індивідуальних структур і об'єднань. Це означає, що вони повинні бути унікальними в межах кожної конкретної структури чи об'єднання, однак можуть збігатися з іменами, що належать іншим класам чи різним структурам і об'єднанням.

4. Останній і самий нечисельний клас утворюють імена міток інструкцій програми. Вони не повинні збігатися між собою в межах одного програмного компонента (функції), однак не зобов'язані відрізнятися від імен, що належать іншим класам, визначених у різних програмних одиницях.