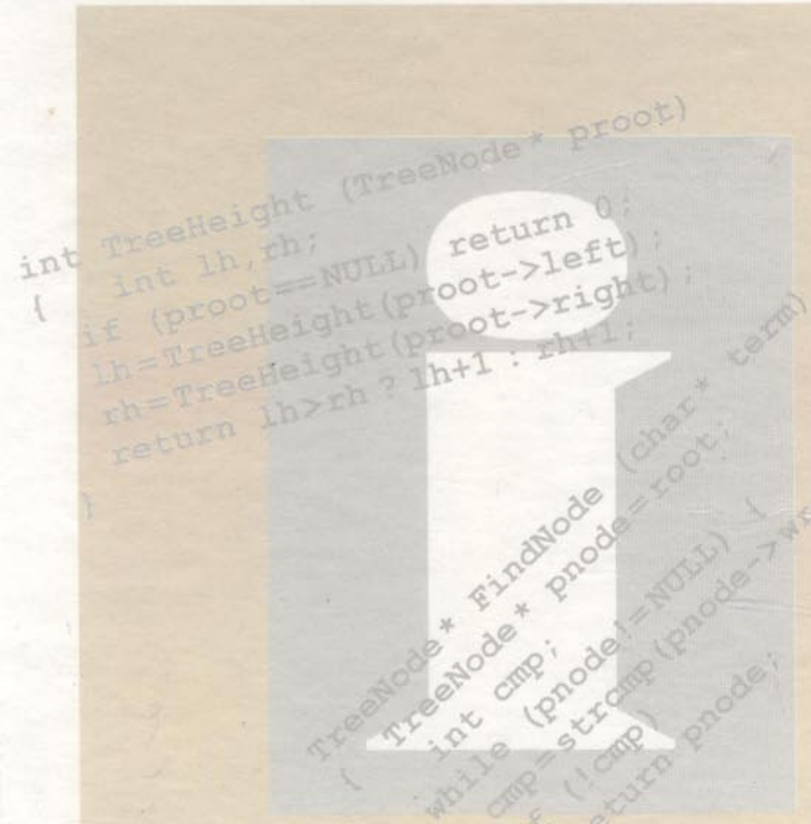
З. Я. Шпак

ПРОГРАМУВАННЯ МОВОЮ C



З. Я. Шпак

ПРОГРАМУВАННЯ МОВОЮ C

Рекомендовано Міністерством освіти і науки України як навчальний посібник для студентів вищих навчальних закладів

Львів
"Оріяна-Нова"
2006

ББК 32.973.2-018.492
Ш 83

Рекомендовано Міністерством освіти і науки України
(Лист № 14/18.2-791 від 28.03.2006 р.)

Видано за рахунок державних коштів. Продаж заборонено

Рецензенти:

Грищук В.В. – член-кореспондент НАН України, доктор технічних наук, професор, директор Державного науково-дослідного інституту інформаційних інфраструктур;

Пасічник В.В. – доктор технічних наук, професор, завідувач кафедри інформаційних систем і мереж Національного університету "Львівська політехніка";

Савакко М.С. – доктор фізико-математичних наук, професор кафедри інформаційних систем у менеджменті Львівського Національного університету ім. І. Франка.

Шпак З.Я.

Ш 83 Програмування мовою С. – Львів: Оріяна-Нова, 2006. – 432 с.: іл.

ISBN 5-8326-0155-6

У навчальному посібнику розглянуто різні аспекти програмування мовою С. У початкових розділах грунтовно описано синтаксис та семантику всіх стандартних конструктивних компонентів мови: лексем, виразів, операторів, функцій. Значну увагу приділено різновидам типів даних, зокрема опрацюванню масивів, символьних рядків, структур тощо. Другу половину посібника присвячено програмним реалізаціям практичних задач, серед яких створення і використання лінійних динамічних списків і двійкових дерев, організація файлового обміну даними, формування екранних зображень та інші.

Викладений матеріал базується на стандарті ANSI/ISO мови С, водночас зазначено нововведення, затверджені в стандарті С-99. Наведено важливу для практичного програмування інформацію про додаткові можливості компілятора, середовища та бібліотек Borland C/C++.

Для студентів, які вивчають програмування в рамках різних навчальних дисциплін, а також для всіх, хто бажає самостійно опанувати мову С.

ББК 32.973.2-018.492

ISBN 5-8326-0155-6

ЗМІСТ

ПЕРЕДМОВА

Мова програмування С, в якій поєднуються потужність і гнучкість універсальних мов програмування з високою ефективністю виконавчого коду та можливістю безпосереднього доступу до апаратних ресурсів комп'ютера, є однією з фундаментальних і найбільш уживаних мов проблемно-орієнтованого та системного програмування. Тому глибоке знання і практичне володіння інструментальними засобами мови С обов'язкове для фахівців з програмного забезпечення комп'ютерів, комп'ютерних інформаційних технологій, систем автоматизованого керування й проектування, комп'ютерної інженерії, а також для всіх, хто пов'язує свою діяльність з комп'ютером і бажає опанувати науку програмування.

Створення та розвиток мови С. Мову С розробив на початку 70-х років минулого століття Деніс Рітчі (*Dennis M. Ritchie*), співробітник фірми Bell Telephone Laboratories (США). Спочатку мова створювалась як ефективний і зручний професійний інструментарій, призначений для програмування операційної системи UNIX. Попередницями С були мови BCPL та B. Довший час фактичним стандартом мови С слугувала її реалізація в 7-й версії UNIX, описана Б. Керніганом (*Brian W. Kernigkhan*) та Д. Рітчі в першому виданні відомої усім програмістам книги "The C Programming Language" (1978 р.). Цю версію мови прийнято позначати K&R.

Властивості мови С настільки захопили програмістів, що незабаром її стали широко використовувати для створення програм у різних практичних сферах. З'явилося багато версій і діалектів мови, часто несумісних між собою. Тому в 1983 р. був сформований комітет із розроблення стандарту мови С. Шість років створювався стандарт С, який у 1989 р. затвердив Американський національний інститут стандартизації (ANSI – American National Standards Institute), а в наступному році – Міжнародна організація стандартизації (ISO – International Standards Organization). Цей стандарт називають ANSI/ISO C, його повністю підтримує переважна більшість сучасних компіляторів мови С.

З 80-х років мовою С розробляють програми практично для всіх типів комп'ютерів: великих універсальних ЕОМ (мейнфреймів), міні-ЕОМ та персональних комп'ютерів, включаючи IBM-сумісні та комп'ютери Macintosh, а також для різних операційних середовищ: MS DOS, UNIX, Windows, Linux, Mac OS, OS/2 та інших. Створюються *системи програмування* С, до складу яких входять *бібліотеки* з широким набором різнотипних функцій та *інтегровані середовища розробки* (IDE – Integrated Development

Enviropment). IDE призначені для швидкого та наочного запису й редагування текстів програм, їх компіляції та налагодження, а також для компонування великих програмних проектів. Одним із найбільш популярних IDE визнано Borland C/C++ 3.1 фірми Borland International (тепер Inprise), розроблене для створення програмних продуктів мовами С та С++ в операційних середовищах MS DOS та Windows.

У 1999 р. затверджено новий стандарт С, який не змінив базових концепцій та конструкцій мови, а лише частково доповнив їх і розширив стандартні бібліотеки з урахуванням нових апаратних і системних можливостей комп'ютерів. На жаль, компілятори С-99 та відповідні середовища програмування ще не набули належного поширення.

Мова С стала основою створення і розвитку низки мов об'єктно-орієнтованого програмування, зокрема: С++, Java, C#. Першою назвою мови С++ була "С з класами", чим підкреслювалось, що С становить фундамент нової мови, а саму мову С навіть оголосили підмножиною С++, проте згодом їх визнали двома самостійними мовами. Насправді, мова С++ є розширенням мови С, оскільки компілятори С++ підтримують усі синтаксичні конструкції та властивості мови С. Додатково С++ включає спеціальні засоби та бібліотеки, які реалізують принципи об'єктно-орієнтованого програмування.

Загальна характеристика мови С. Сучасна універсальна мова програмування С призначена для створення прикладних (ужиткових) програмних продуктів і системних компонентів програмного забезпечення комп'ютерів. Підкреслимо основні риси мови С.

Потужність і гнучкість – ці терміни вживають найчастіше, характеризуючи С. Мовою С написано більшу частину програм універсальної і потужної операційної системи UNIX, створено ряд компіляторів та інтерпретаторів з різних мов програмування (зокрема Pascal, APL, LISP, Basic тощо), розроблено велику кількість інтерфейсів, різноманітних інструментальних засобів, в т. ч. текстових і графічних редакторів, систем і спеціалізованих програм для наукових досліджень, комп'ютерних ігор та багато іншого.

Ефективність. Програми мовою С достатньо швидкодійні й економні за обсягами оперативної пам'яті, оскільки вони базуються на реальних можливостях комп'ютера. Це пов'язане з тим, що С, хоча і є універсальною мовою програмування, але не належить до мов високого рівня, як, наприклад, Pascal, Ada, Modula-2 чи Basic. С – мова середнього рівня, яка може оперувати безпосередньо з даними та їх елементами: байтами, бітами, словами, адресами. Цим С-програми подібні до програм, написаних мовою асемблера, їх можна налаштовувати або на максимальну швидкодію, або на оптимальне використання оперативної пам'яті.

Структурованість. С належить до мов, які реалізують принципи структурного підходу, зокрема проектування програм "зверху вниз", модульність, локалізацію області дії імен, автономність підпрограм, відокремлення коду програми від даних та інші. Структурованість С-програм близька до структурованості програм, написаних мовами Pascal чи Modula-2. Натомість С використовує єдиний вид підпрограм – функції. Мова С підтримує практично всі керуючі конструкції, а також типи і структури даних, які визнані сучасною теорією та практикою програмування як найбільш ефективні.

Орієнтація на професіоналізм програміста. Одна з базових концепцій С – високий рівень довіри до програміста. Створена для потреб програмістів-професіоналів, мова С надає користувачеві широку свободу вибору форм даних і засобів програмування. Водночас потрібно пам'ятати, що відсутність контролю може призвести до небезпечних помилок у роботі програми, тому на програміста лягає значно вища, ніж в інших мовах, відповідальність за правильність результатів виконання програми.

Лаконізм. Ще однією рисою С, яка приваблює програмістів, є лаконізм запису конструктивних компонентів програм, пов'язаний передусім з невеликою кількістю ключових слів і потужним набором операцій. Програми мовою С можуть бути гранично компактними і короткими. Проте інколи надмірне прагнення лаконізму робить програмний код малозрозумілим і сприятливим для помилок, які важко виявити. Недарма для С-програм традиційно проводять конкурс на найбільш заплутану програму.

Мобільність. Важливою рисою сучасних мов програмування є їх мобільність, яка дає змогу переносити програму, написану в одній системі програмування (тобто на певній операційній платформі) в інше операційне середовище. Мова С належить до мов, які характеризуються доброю мобільністю: перенесення програми вимагає мінімальних змін або доповнень. Безумовно, ті програми чи програмні компоненти, що використовують не стандартні засоби мови, а орієнтуються на конкретне апаратне забезпечення (наприклад, на певний вид відеосистеми або особливості механізму звертання до файлів), залишаються немобільними.

Підсумовуючи сказане, можна зробити загальний висновок: С – це універсальна мова програмування середнього рівня, яка підтримує більшість сучасних концепцій програмування, характеризується достатньою потужністю, доброю структурованістю, високою ефективністю, можливістю безпосереднього доступу до даних, компактністю та мобільністю програм. Основним недоліком мови є низька надійність через широку свободу і неконтрольованість дій програміста.

Слово до читача. Щоб навчитись програмувати, недостатньо ознайомитись з мовою чи перегляпути певну кількість програм. Необхідно самостійно складати та реалізовувати програми, перевіряючи їх працездатність за різних умов виконання, шукати оптимальні програмні рішення й експериментувати, вдосконалюючи зміст і стиль запису програм. Потужна за своїми можливостями і водночас не надто складна, лаконічна й елегантна мова С, яка відповідає більшості сучасних вимог до мов практичного програмування, дає змогу всебічно оволодіти майстерністю створення програм. "... чим більше працюєш із мовою С, тим зручнішою вона стає," – зазначали у передмові до своєї книги Б. Керніган і Д. Рітчі. Цю тезу можуть підтвердити тисячі програмістів у всьому світі.

Запропонований навчальний посібник містить достатньо повний опис мови С, який побудовано за принципом "від простого до складного". Спочатку розглянуто синтаксис і семантику базових конструктивних елементів мови: лексем, виразів та операторів. Детально описані всі різновиди типів даних, як простих (арифметичних і вказівникових), так і складених (масивів, структур, об'єднань). Значну увагу приділено програмуванню функцій, які є основними структурними компонентами С-програм. Матеріал подано так, щоб його можна було вивчати в довільному порядку.

Останні розділи посібника присвячено питанням практичного програмування задач, пов'язаних із опрацюванням числової та текстової інформації. Обґрунтовується застосування динамічної пам'яті для збереження даних і формування таких інформаційних структур, як одно- й двозв'язні списки та двійкові дерева. Розглядаються засоби організації роботи з файлами та керування консольним введенням/виведенням даних.

Викладений матеріал проілюстровано великою кількістю програмних прикладів, які детально роз'яснені та підібрані так, щоб розкрити основні прийоми, алгоритми та методи програмування. Всі програми протестовано в середовищі Borland C/C++ 3.1. Важливими є практичні поради, які допоможуть початківцям уникнути характерних "прихованих" помилок у процесі створення власних програм. Наприкінці кожного розділу подано контрольні запитання та завдання, що дає змогу закріпити набуті знання.

Умовні найменування використані в посібнику:

- C-89 – ANSI/ISO стандарт мови С, затверджений у 1989/1990 рр. Основний матеріал базується саме на цьому стандарті, оскільки він реалізований практично в усіх середовищах програмування мови С, а також найбільш сумісний за синтаксисом і семантикою з мовою C++;
- C-99 – новий ISO/IEC стандарт мови С, затверджений у 1999 р. У відповідних місцях тексту розглядаються зміни, доповнення чи розширення, внесені в мову С цим стандартом. Перелік основних нововведень, запроваджених стандартом С-99, подано в Додатку 3;
- Borland C – умовне ім'я, яким позначатимемо ту частину системи програмування (компілятор, бібліотеки, IDE тощо), яка застосовується для створення С-програм у середовищі Borland C/C++. Це середовище найчастіше застосовують для реалізації програм, написаних мовою С. Тому крім стандартних засобів С у посібнику описані особливості та додаткові можливості, якими характеризується Borland C.

Умовні позначення:

 – слід обов'язково звернути увагу;

 – допоміжна інформація;

 – контрольні запитання та завдання.

Розділ 1

БАЗОВІ ЕЛЕМЕНТИ МОВИ

У цьому розділі:

- Алфавіт мови С
- Характеристика ASCII-кодів символів
- Керуючі ескейп-послідовності
- Лексеми С: ключові слова, ідентифікатори, константи, символьні рядки, знаки операцій, роздільники, коментарі

Вивчення мови розпочнемо з розгляду її найменших структурних елементів – алфавіту та лексем, з яких формують всі конструктивні компоненти С-програм: декларації типів, описи змінних, вирази, оператори, функції та інші.

1.1. Алфавіт

Алфавіт мови визначає набір символів, які можуть використовуватись для формування лексичних елементів програм.

Всі символи алфавіту С поділяють на три групи. До першої групи входять символи ключових слів та ідентифікаторів, до яких належать:

- великі та малі літери латинської абетки: `A..Z, a..z` (всього 52 символи);
- цифри: `0..9` (всього 10 символів);
- знак підкреслення: `_` .

До другої групи входять символи, що використовуються як знаки операцій, символи пунктуації та роздільники:

```
+   -   *   /   %   =   <   >   &   |   !   ~   ^   ?   ,   .
;   :   '   "   (   )   [   ]   {   }   #   \   (всього 28 символів).
```

Третя група символів С – це т. зв. неграфічні символи, кожен з яких має встановлений внутрішній код, як і символи першої та другої груп, але не має власного графічного позначення. До цієї групи належать символ пробілу та спеціальні керуючі символи, які ще називають ескейп-послідовностями: символи табуляції, нового рядка, нової сторінки тощо. Пробільні неграфічні символи використовують у С-програмах для відокремлення лексем.

Таблиця ASCII-кодів символів. В оперативній пам'яті комп'ютера кожен символ зберігається як ціле двійкове число, значення якого відповідає коду цього символа. Найбільш поширеною системою кодування символів є таблиця ASCII-кодів (ASCII – скорочено від American Standard Code for Information Interchange – американський стандартизований код для обміну інформацією). Згідно з цією системою кодування код символа є однобайтовим, що дає змогу занести в повну ASCII-таблицю 256 символів з кодами від 0 до 255.

Перша половина ASCII-таблиці (символи з кодами від 0 до 127) є строго стандартизованою і становить основу ASCII-системи кодування. В цій частині таблиці містяться коди всіх символів, що складають алфавіт мови С. Зокрема, код символа * дорівнює 42, символа 1 – 49, символа = – 61, символа B – 66, а символа b – 98 і т. д. Перші 33 символи кодової таблиці (з кодами від 0 до 32) належать до неграфічних – вони не мають встановленого символьного зображення і призначені для виконання спеціальних функцій. Наприклад, символ з кодом 9 задає горизонтальний табуляційний відступ, символ з кодом 27 позначає клавішу *Esc*, значенням 32 кодується символ пробілу тощо. Повну таблицю стандартних ASCII-кодів наведено в Додатку 1.

Друга половина ASCII-таблиці (символи з кодами від 128 до 255) використовується для кодування додаткових символів, у тому числі символів т. зв. псевдографіки, які застосовують для малювання рамок таблиць і вікон, а також для створення простих рисунків у текстових режимах роботи відеосистеми. У другу половину ASCII-таблиці заносять символи національних алфавітів, власне там розміщені коди літер кирилиці. Цю частину кодової таблиці називають розширеною, її вміст може змінюватись, передусім, через особливості кодування національних алфавітів. У Додатку 1 подано дві розширені ASCII-таблиці з кодами літер української та російської абеток, перша з яких є базовою для операційної системи MS DOS, а друга – для Windows.

Звернемо увагу на деякі властивості кодування символів у ASCII-таблицях, що мають практичне застосування у програмуванні багатьох задач:

- коди символів цифр 0 .. 9 розташовані підряд у порядку зростання: 48 .. 57;
- коди великих латинських літер A .. Z та малих літер a .. z теж записані послідовно в абетковому порядку: великі літери – 65 .. 90, малі – 97 .. 122;
- різниця кодів малих і відповідних великих латинських літер однакова та дорівнює 32;
- символи псевдографіки в розширеній ASCII-таблиці мають коди від 176 до 223;
- кодування літер кирилиці для двох основних кодових таблиць: таблиці кодів MS DOS і таблиці кодів Windows є різним (див. табл. Д1.3 і Д1.4 у Додатку 1);
- в обох таблицях послідовність кодів літер кирилиці не є неперервною, крім того, порядок кодів не відповідає строго порядку літер в українській абетці;
- різниця кодів малих і відповідних великих українських літер не однакова для різних пар літер.

Про перелічені особливості кодування символів, насамперед латинських і українських літер, треба пам'ятати під час програмування задач, пов'язаних із перетвореннями мала ⇔ велика літера та порівнянням чи впорядковуванням символьних рядків.

Спеціальні керуючі символи. Ці символи називають *керуючими послідовностями* або *ескейп-послідовностями* (*Esc-послідовностями*). Вони позначають неграфічні символи, призначені, для керування формою виведення даних і повідомлень: відтворення короткого звукового сигналу, повернення екранного курсора на одну позицію назад, переведення курсора на початок рядка тощо (табл. 1.1).

Ескейп-послідовності у С-програмах можна записувати трьома способами, використовуючи символьну форму або вісімкове чи шістнадцяткове позначення. У всіх трьох варіантах запису першим символом послідовності є зворотна коса риска (її називають лівим слешем). У символьній формі ескейп-послідовності за слешем записується визначена

для даного символа літера, наприклад, `\t` позначає перехід до наступної горизонтальної табуляційної позиції. Вісімкове позначення ескейп-послідовності складається зі слеша та однієї, двох або трьох вісімкових цифр (цифри від `0` до `7`), що задають код даного символа. Зокрема, ескейп-символ `\t`, ASCII-код якого дорівнює 9, можна записати в формі `\011` або `\11`. Шістнадцяткова форма ескейп-послідовності починається символами `\x` або `\X`, за якими вказуються одна або дві шістнадцяткові цифри (цифри `0..9` та літери `a..f` або `A..F`). Шістнадцяткове позначення `\t` може бути таким: `\x09` або таким: `\X9`.

Таблиця 1.1

Символи керуючих послідовностей

Слеш-символ	Призначення символа	ASCII-код	Вісімкове позначення	Шістнадцяткове позначення
`\a`	звуковий сигнал	7	`\007`	`\x07`
`\b`	повернення на символ	8	`\010`	`\x08`
`\t`	горизонтальна табуляція	9	`\011`	`\x09`
`\n`	перехід на новий рядок	10	`\012`	`\x0a`
`\v`	вертикальна табуляція	11	`\013`	`\x0b`
`\f`	перехід до нової сторінки	12	`\014`	`\x0c`
`\r`	перехід на початок рядка ("повернення каретки")	13	`\015`	`\x0d`

Як ескейп-послідовності записують також вказані нижче символи абетки, коли вони використовуються як окремі символьні константи або як елементи символьних рядків:

`\'` – позначення апострофа;
`\"` – позначення лапок;
`\\` – позначення лівого слеша;
`\?` – позначення знака запитання.

Слід запам'ятати ці символи, бо відсутність лівого слеша в їх записах може викликати неправильну інтерпретацію компілятором символьних рядків, тим самим спричинити помилковість результатів роботи програми.

У Borland C лівий слеш для символа `?` не обов'язковий.

Якщо лівий слеш записати перед будь-яким іншим символом, що не входить у наведений вище перелік керуючих або спеціальних символів, то такий слеш компілятором просто ігнорується. Тобто, двосимвольна послідовність `\m` інтерпретується як символ `m`, а послідовність `\#` – як звичайний символ `#` тощо.

Через вісімкову та шістнадцяткову форму ескейп-послідовностей можна записати довільний символ ASCII-таблиці. Наприклад, ескейп-послідовність `\0` позначає символ з кодом `0` – перший символ ASCII-таблиці. Цей символ використовують у символьних рядках як ознаку кінця рядка. Ескейп-послідовність `\X1B` (або рівнозначна вісімкова

форма \033) відповідає символу ASCII-таблиці з кодом 27, що позначає клавішу *Esc*, а ескейп-послідовність \xBA (або \272) задає символ псевдографіки з кодом 186 – подвійну вертикальну лінію ‖.

Незважаючи на те, що в записах ескейп-послідовностей використовуються два, три або й чотири символи, всі вони позначають один символ стандартної кодової таблиці й розглядаються компілятором нарівні з іншими символами мови.

1.2. Лексеми

Найменшими змістовними елементами програм, що мають самостійне призначення, є *лексичні одиниці* (*лексеми*). Лексеми мови С поділяють на шість груп: ключові слова, ідентифікатори, константи, символьні рядки, знаки операцій та роздільники.

Ключові слова. *Ключові слова* (їх ще називають *службовими* або *зарезервованими* словами) – це набір визначених слів, що використовуються для встановлення типів даних, класів пам'яті даних, формування операторів тощо. Кожне ключове слово має своє призначення, застосовувати ключові слова для іншої мети (зокрема як імена змінних чи функцій) заборонено.

Мова С використовує невеликий набір ключових слів – їх усього 32. Перелічимо ці слова, поділивши їх на групи за призначенням:

- ключові слова типів даних:

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

- ключові слова кваліфікаторів типів і специфікаторів класів пам'яті:

```
auto   const   extern   register   static   volatile
```

- ключові слова операторів:

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

- інші ключові слова:

`sizeof` – операція, `typedef` – декларація типу.

Стандарт С-99 ввів у мову ще п'ять ключових слів:

```
_Bool   _Complex   _Imaginary   inline   restrict
```

Крім ключових слів, що підтримуються стандартом мови С, ряд компіляторів використовує додаткові ключові слова, призначені для взаємодії з програмами, написаними іншими мовами програмування, а також для врахування апаратних чи операційних особливостей середовища, в якому реалізується програма. Зокрема, у середовищі Borland С додатково зафіксовані як ключові наступні слова:

```
asm   cdecl   _cs   _ds   _es   far   fortran   huge   interrupt
near   pascal   _ss
```

Звернемо увагу на те, що в ключових словах великі та малі літери вважаються різними. Тому в програмах ці слова треба записувати так, як вони вказані вище, наприклад, `int`, а не `Int` чи `INT`.

Ідентифікатори. Імена змінних, макросів, міток, функцій та інших об'єктів програми називають *ідентифікаторами*. Ідентифікатори формують із символів першої групи, тобто з малих і великих латинських літер, цифр, а також знака підкреслення.

Приклади коректних і помилкових ідентифікаторів:

- `summa` `x1` `mitka_3` `PrintList` `new_array` – правильні;
- `2root` – неправильно, починається з цифри;
- `res'05` – неправильно, містить символ апострофа;
- `Clear Heap` – неправильно, містить символ пропуску.

Для наочності та зрозумілості програми важливу роль відіграє змістовність імен об'єктів. Приміром, ідентифікатор змінної `file_name` відразу вказує, що ця змінна пов'язана з іменем файла, а ім'я функції `OpenNewWindow` асоціюється з процедурою відкриття нового вікна. Такої наочності не буде, якщо для цих об'єктів використати короткі ідентифікатори, наприклад, `fn` та `win`.

Як і в ключових словах, в ідентифікаторах розрізняються великі та малі літери. Зокрема, три наступних ідентифікатори: `prod`, `Prod` та `PrOd` будуть розглядатись компілятором як різні імена.

Прийнято, що імена внутрішніх об'єктів системи програмування, які використовуються в реалізаціях компіляторів і бібліотечних функцій, починаються одним або двома знаками підкреслення: `_heaplen` чи `__FILE__`. Тому не рекомендується починати зі знака підкреслення імена змінних та інших об'єктів програми, щоб випадково не спричинити конфлікту імен.

Константи. *Константи* (у літературі можна зустріти також термін *літерали*) – це об'єкти програм, значення яких змінювати не можна. У мові С константи належать до арифметичних даних. Стандарт мови поділяє константи на цілочислові, дійсні, символьні та перелікові. Питання синтаксису констант, встановлення їх типів, практичного використання та інші детально розглядаються в розділі 3, тут наведемо лише коротку загальну характеристику кожної групи констант.

Цілочислові константи використовуються для позначення цілих чисел зі знаком або без знака. Можуть мати три форми зображення:

- *десяткові* константи: `5246` `200` `-13` `+45`
- *вісімкові* константи: `034` `01002` `0515` `0777`
- *шістнадцяткові* константи: `0x9243` `0XDA07` `0xa3` `0X10045F`

Вісімкові константи завжди починаються цифрою `0`, за якою записуються вісімкові цифри значення константи (цифри від `0` до `7`). Шістнадцяткові константи починаються префіксом `0x` або `0X` (нуль і маленька чи велика літера `x`), за яким вказуються шістнадцяткові цифри числа (цифри `0..9` та літери `a..f` або `A..F`).

Дійсні константи застосовують для позначення чисел, які мають як цілу, так і дробову частину (тобто дійсних чисел), а також великих цілих чисел. Дійсні константи можна записувати у двох формах:

- константи з *фіксованою крапкою*: `7.123` `0.060` `-384.65`
- константи з *плаваючою крапкою*: `276.8e-5` `10.02e+8` `5157E10`

Константи з фіксованою крапкою містять цілу і дробову частини числа, які відокремлюються між собою десятковою крапкою. У константах з плаваючою крапкою додатково вказується десятковий порядок числа, який записується після літери `e` або `E` (від слова *експонента*). Наприклад, константа `276.8e-5` відповідає дійсному числу $276{,}8 \cdot 10^{-5}$. Це число можна записати інакше: `2.768e-3` або `0.2768e-2`, а також через константу з фіксованою крапкою `0.002768`.

Символьні константи (їх ще називають *літерними*) використовуються у програмах для звертання до окремих символів. Символьна константа позначається одним символом, записаним у апострофах, або ескейп-послідовністю, теж записаною в апострофах. У символьних константах можна використовувати всі символи активної кодової таблиці, зокрема, літери кирилиці, якщо вони наявні в цій таблиці. Приклади символьних констант:

```
'A'  'a'  '*'  '5'  'Я'  'щ'  '|'  '\n'  '\a'  '\''  '\XB2'
```

Символьні рядки. Ще одним видом лексем мови С є символьні рядки (у літературі їх також називають *стрінговими літералами* або *рядковими* (*стрінговими*) *константами*). *Символьний рядок* – це послідовність довільних символів кодової таблиці (серед них можуть бути ескейп-послідовності), охоплена лапками:

```
"Press any key..."
"Національний університет \"Львівська політехніка\""
"\t Розв\'язок: \n"
```

Зверніть увагу на ескейп-послідовності, якими в другому і третьому рядках позначено внутрішні лапки, апостроф і керуючі символи.

В оперативній пам'яті всі символи рядка розташовуються підряд. Кожен символ, включаючи ескейп-послідовності, займає один байт, в якому записується код символа (зазначимо, що деякі системи використовують двобайтове кодування символів).

Особливість символьних рядків мови С в тому, що компілятор автоматично долучає до них кінцевий *нуль-символ* `'\0'`, який використовується у процесах опрацювання рядків як ознака їхнього кінця. Два наступні записи: `'C'` і `"C"` є різними не тільки синтаксично, а й семантично. Запис `'C'` позначає символьну константу, яка зберігається у пам'яті як ціле число, значення якого дорівнює коду літери `C`. У той час як запис `"C"` є символьним рядком, що складається з двох символів: літери `C` та нуль-символа (`'C'+'\0'`).

Довжина символьного рядка не обмежується. У разі довгих стрінгових констант часто виникає потреба запису їх у декількох рядках програми. Використовують два способи

поділу символьних рядків. Перший полягає в тому, що в місці розриву рядка записують лівий слеш, за яким ставлять символ нового рядка (натискують клавішу *Enter*). Тоді записані в наступному рядку символи вважаються продовженням стрінгової константи:

```
"Приклад довгого символьного рядка \
        з перенесенням"
```

Записаний вище рядок є рівнозначним до наступного:

```
"Приклад довгого символьного рядка        з перенесенням"
```

Недолік цього способу в тому, що в рядок результату потрапляють всі символи пробілу, записані перед символом слеша та на початку нового рядка.

У другому способі перенесення стрінгів використовується та властивість, що компілятор об'єднує (*конкатенує*) у спільний рядок два записані підряд стрінги – між ними може бути довільна кількість символів-роздільників: пробілів, символів нового рядка, табуляції тощо. Тому довгий символьний рядок можна просто поділити на частини:

```
"Ще один приклад перенесення"
        " довгого символьного рядка"
```

Відповідний повний рядок буде таким:

```
"Ще один приклад перенесення довгого символьного рядка"
```

Знаки операцій, роздільники, коментарі. Описані вище лексеми: ключові слова, ідентифікатори та константи – це т. зв. *лексеми-слова*. У програмі між двома лексемами-словами обов'язково має бути записаний *знак операції* або *роздільник*: знак пунктуації чи пробільний символ.

Знаки операцій можуть позначатись одним символом або дво- чи багатосимвольною комбінацією зі символів другої групи алфавіту мови С. Приклади знаків операцій:

```
+   *   &   <   .   ^   =        – односимвольні операції;
++  ||  >>  ->  /=  <<=          – багатосимвольні операції.
```

Детальному розгляду операцій мови С присвячено розділ 4.

До роздільників належать символи, які називають *знаками пунктуації*:

```
( )   [ ]   { }   ,   ;   :   =   *   #
```

Зокрема, фігурними дужками { } оформлюють тіло функції, знаком ; завершують усі описи та оператори, через знак = ініціалізують змінні в оголошеннях, "зірочкою" * відзначають в описах вказівники тощо.

Один і той же символ у мові С може мати різне призначення залежно від контексту, в якому він використовується. Наприклад, знаком * позначають дві операції: арифметичне множення і звертання до даних за їх адресами, а також застосовують його як знак пунктуації для оголошення даних вказівникового типу.

Роль роздільників лексем відіграють також символи, які називають *пробільними*: пробіл, символи горизонтальної і вертикальної табуляції, нового рядка, нової сторінки, переходу на початок рядка. Ці символи можна записувати в довільній кількості

між будь-якими двома лексемами – самі ж лексеми розривати не можна. Компілятор розглядає послідовність довільних пробільних символів як один розділовий символ, що відокремлює лексеми.

У цьому ж призначенні роздільником є і *коментар* – текст, що роз'яснює програму. Кожен коментар починається двосимвольною комбінацією `/*` і закінчується парою символів `*/`. Приклад коментаря:

```
/*  Сортування рядків за порядком української абетки  */
```

Коментарі можна записувати в довільному місці програми між лексемами. Додаткову інформацію про коментарі можна прочитати в наступному розділі.

Запитання для самоконтролю

1. Чи є на основній клавіатурі персонального комп'ютера символи, що не входять до алфавіту мови С?
2. Відомо, що ASCII-код символа цифри `0` має значення 48. Яким є ASCII-код цифри `5`?
3. За розширеними ASCII-таблицями знайдіть і порівняйте, як кодуються українські літери Ж та ж, І та і, а також Ї та ї у системі кодування, яка застосовується в операційному середовищі MS DOS та в системі кодування, яку підтримує середовище Windows.
4. Яке призначення кожної з наведених ескейп-послідовностей: `\07`, `\n`, `\\`, `\X0D`, `\xdc`?
5. Що таке ключові (службові) слова мови? Назвіть декілька ключових слів мови С. Яке їх призначення?
6. Чи всі записані далі ідентифікатори є правильними: `sto`, `st3`, `1prop` `grup-2`, `grup_5`, `result#4`?
7. Поділіть перелічені цілочислові константи на десяткові, вісімкові та шістнадцяткові: `1101`, `01101`, `792`, `0`, `0X1101`, `0x7`, `07`, `7`.
8. Які константи належать до дійсних? Наведіть приклади дійсних констант.
9. Чи однаковими є дві наступні константи: `'N'` та `"N"`? До яких груп вони належать?
10. Чим можуть бути відокремлені лексеми-слова в тексті програми?
11. Які символи можна використовувати в символьних рядках і коментарях?
12. Як можна переносити на наступний рядок довгі стрінгові константи?

Розділ 2

СТРУКТУРА С-ПРОГРАМИ. ВИКОНАННЯ ПРОГРАМ

У цьому розділі:

- Перші ознайомчі програми мовою С
- Головна функція С-програм – `main()`
- Структура програми, описи даних, оператори, користувацькі функції
- Бібліотечні функції та заголовні файли, директиви препроцесора, макропідстановки
- Стиль запису та оформлення С-програм, коментарі
- Етапи виконання програм: введення і редагування тексту, препроцесування і компіляція, компонування виконавчого коду

У цьому розділі розпочнемо ознайомлення зі структурою програм, написаних мовою С, та їх основними компонентами. Вивчаючи наступні теми, будемо поглиблювати й розширювати знання про *синтаксис* (правила, які визначають вид і форму записів) та *семантику* (змістовне значення) конструктивних компонентів С-програм і структури програми в цілому.

2.1. Приклад короткої програми

Розпочнемо з наступної, гранично короткої, але цілком працездатної програми:

```
/*  Виведення текстового повідомлення  */
#include <stdio.h>
void main(void)
{
   puts ("\n Це наша перша С-програма.");
}
```

У разі запуску цієї програми на виконання на екран буде виведено рядок тексту:

```
Це наша перша С-програма.
```

Подальші пояснення допоможуть зрозуміти структуру наведеної програми.

Функції С-програм. Основними складовими частинами кожної С-програми, якою б короткою чи великою за обсягом вона не була, є функції. *Функція* – це синтаксично та логічно завершений самостійний фрагмент програми, що має власне ім'я і реалізує певну задачу. У програмах використовують два види функцій: *бібліотечні*, які вже створені й зберігаються в бібліотеці системи програмування, та *користувацькі*, які створює програміст у даній програмі. Наша перша програма містить єдину користувацьку функцію `main()`, в якій відбувається звертання до стандартної бібліотечної функції `puts()`.

Для підвищення наочності й читабельності програм імена користувацьких функцій у текстах програм будемо виділяти напівжирним шрифтом.

Всі функції мови С мають однакову структуру:

```
тип_значення_функції ім'я_функції (список параметрів)
{
    тіло функції
}
```

Початкова частина функції, яка складається з типу значення функції, її імені та списку оголошень параметрів функції, записаного в круглих дужках, називається *заголовком функції*. *Тіло функції* охоплюється фігурними дужками і формується з описів даних та операторів, що реалізують дії, які повинна виконувати дана функція.

Функція `main()`. Користувацькі функції можуть мати довільні імена, які надає їм програміст. Водночас у програмі обов'язково повинна бути функція з іменем `main` (головна) – з неї починається виконання кожної програми, а закінчення `main()` викликає завершення роботи всієї програми.

Звернемо увагу на заголовок функції `main()`. У наведеній вище короткій програмі перше слово `void` є типом значення, яке повертає функція. Це значення після завершення роботи `main()` передається у програму операційної системи, яка запустила на виконання дану користувацьку програму. Borland C допускає, щоб у програмах, значення завершення яких не треба передавати в операційне середовище, *тип_значення_функції* в заголовку функції `main()` оголошувати ключовим словом `void` (вільний, порожній). Слово `void` зазначає, що функція не повертає значення. Проте деякі операційні середовища вимагають, щоб кожна програма повертала ціле число, яке вказує, яким чином завершено роботу цієї програми: нормально (коректно) чи аварійно (помилково). Тому надалі, для дотримання стандарту мови С, будемо встановлювати тип значення функції `main()` як `int` (цілий), а перед закінченням функції повертатимемо значення `0`, що сигналізує про безпомилкове завершення програми.

```
/*  Стандартизований варіант попередньої програми  */
#include <stdio.h>
int main(void)
{
  puts ("\n Це наша перша С-програма.");
  return 0;
}
```

Слово `void` у списку параметрів функції `main()` підкреслює, що дана функція не використовує ніяких параметрів.

У старих книгах з програмування мовою С можна зустріти програми, в яких перед словом `main` та в списку параметрів функції нічого не вказано:

```
main()
{
  . . .          /* тіло програми */
  return 0;
}
```

Згідно з першими стандартами такі оголошення функцій використовували т. зв. правила замовчування, за якими вважалось, що значення функції має тип `int`, а список параметрів невизначений. Стандарт С-99 скасував правило встановлення типу `int` за замовчуванням і вимагає явного вказання типу для значення, яке повертає функція. Такі ж вимоги щодо оголошення функцій і в мові С++. У разі порожнього списку параметрів компілятор С не контролює відповідності параметрів, тому, щоб уникнути помилок, відсутність параметрів функції слід позначати словом `void`.

Тіло функції формують з описів змінних і операторів. Оскільки в наших перших коротких програмах змінні не використовувались, то про них поговоримо пізніше. *Оператори* – це конструктивні елементи програм, що виконують певні встановлені дії. Робота програми здійснюється саме через оператори. Кожен оператор мови С завершується знаком крапка з комою `;`.

У першому варіанті нашої програми є тільки один оператор:

```
puts ("\n Це наша перша С-програма.");
```

який викликає стандартну бібліотечну функцію `puts()`, призначену для виведення на екран заданого символьного рядка. У другому варіанті програми в функцію `main()` додатково введено прикінцевий оператор

```
return 0;
```

який завершує роботу програми і передає в операційне середовище значення `0`.

Бібліотечні функції та стандартні заголовні файли. Мова С не має вбудованих (таких, що входять до складу компілятора) засобів введення/виведення даних. Тому, якщо в процесі роботи програми потрібно отримати якусь інформацію від користувача чи відобразити результати виконаних операцій, то застосовують бібліотечні функції С, призначені для введення/виведення даних, або створюють власні користувацькі функції. У нашій програмі для відтворення на екрані текстового повідомлення використано стандартну бібліотечну функцію `puts()`.

Компілятор С здійснює контроль правильності звертання до функцій. Для цього кожна функція, що використовується у програмі, повинна бути попередньо описана або оголошена. Оголошення бібліотечних функцій згруповані у т. зв. заголовних файлах, які необхідно під'єднувати до програми через директиву включення файла `#include`. Всі бібліотечні функції стандартизованого введення/виведення даних оголошені в спеціальному заголовному файлі `<stdio.h>`.

Директиву включення заголовного файла <stdio.h>:

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

будемо записувати в кожній програмі, яка використовує стандартні бібліотечні функції високорівневого обміну даними.

Кутові дужки < > в імені заголовного файла вказують, що даний файл зберігається в спеціальному каталозі, виділеному в середовищі програмування для розміщення заголовних файлів. У Borland C цей каталог стандартно має ім'я INCLUDE. Назви заголовних файлів пов'язані з призначенням функцій, які оголошені в цих файлах. Так, назва stdio.h є скороченням від слів *standard input/output header* – заголовок стандартного введення/виведення; назва math.h вказує, що даний заголовний файл зберігає математичні функції; назву conio.h має файл, в якому зібрані функції, що розширюють можливості консольного введення/виведення даних тощо. Всі заголовні файли є текстовими файлами MS DOS, їх можна переглянути, використовуючи довільний редактор текстів. Оскільки спочатку вміст заголовних файлів може бути не зовсім зрозумілим, краще знайомитись з їхнім призначенням через підсистему допомоги Help інтегрованого середовища Borland C (пункт меню Help/Contents/Header Files).

2.2. Структура програми

Щоб детальніше пояснити структуру С-програм, розглянемо ще один приклад. Подана нижче програма виводить на екран таблицю, у N рядках якої вказуються значення температур у градусах Цельсія та відповідні значення у градусах Фаренгейта. Початкове значення температури за Цельсієм вводить користувач, крок зміни температури задається програмно через параметр DELTA_TC. Перетворення значення температури із градусів Цельсія у градуси Фаренгейта виконує окрема функція Cels_to_Fahr(). Для зручності наступних пояснень рядки тексту програми пронумеровано.

```
1  /* Виведення таблиці температур у градусах Цельсія
2     та градусах Фаренгейта */
3  #include <stdio.h>
4  #define N  20                        /* кількість значень  */
5  #define DELTA_TC  1                  /* крок зміни температури */
6  double Cels_to_Fahr (int tC);        /* прототип функції */
7  int main(void)
8  {
9      int n,                  /* лічильник рядків таблиці */
10         tmin,               /* початкове значення температури */
11         tCels;              /* температура в градусах Цельсія */
12     double tFahr;           /* температура в градусах Фаренгейта */
13   printf("\n Початкове значення температури - ");
14   scanf("%d", &tmin);
```

```
15    printf("\n\t Таблиця температур\n tC (за Цельсієм)"
16           "    tF (за Фаренгейтом)\n");
17    for (tCels=tmin, n=1; n<=N; n++) {
18      tFahr=Cels_to_Fahr(tCels);
19      printf("%10d %21.1f \n", tCels, tFahr);
20      tCels+=DELTA_TC;
21    }
22    return 0;
23  }
24  /*  Функція перетворення температур  */
25  double Cels_to_Fahr (int tC)
26  {
27    return (double)tC*1.8+32.0;
28  }
```

Приклад виконання програми:

```
Початкове значення температури -  -5
        Таблиця температур
tC (за Цельсієм)     tF (за Фаренгейтом)
      -5                    23.0
      -4                    24.8
      -3                    26.6
             . . .
      13                    55.4
      14                    57.2
```

Директиви препроцесора. Макропідстановки. Три початкові рядки програми, які починаються символом `#`, містять т. зв. *директиви препроцесора*. Про директиву `#include`, яка під'єднує до програми заголовний файл `<stdio.h>`, мова вже йшла раніше. Цей файл необхідний, оскільки в програмі для введення та виведення даних використовуються відповідні бібліотечні функції `scanf()` та `printf()`.

У рядках 4 і 5 записано директиву `#define`. Ця директива виконує заміну *макроконстанти*, вказаної після слова `define`, на текст підстановки, що записується після імені макроконстанти. Заміна виконується в усьому тексті програми, починаючи з рядка, наступного за даною директивою. Зокрема, в наведеній програмі всі лексеми `N` будуть замінені константою `20`, а лексеми `DELTA_TC` – константою `1`.

Заміни, які виконує директива `#define`, у літературі називають *макропідстановками*, *макрозамінами* або *макророзширеннями*. Саму ж лексему, яка замінюється заданим текстом підстановки, називають *іменованою константою*, *макроконстантою* або просто *макросом*. Детально особливості виконання макропідстановок розглянуті в розділі 17, зараз зупинимось тільки на декількох моментах:

- застосування макроконстант дає дві істотні переваги: по-перше, зростає наочність програми, оскільки ім'я макроса несе інформацію про призначення даної константи (у нашому прикладі `DELTA_TC` – крок зміни значення температури за Цельсієм);

по-друге, спрощується внесення змін у програму – достатньо змінити значення у виразі підстановки, щоб відповідна зміна була виконана всюди в тексті програми;

- ім'я макроконстанти може бути довільним ідентифікатором, але прийнято записувати макроси заголовними літерами, щоб у програмі вирізнити їх з-поміж інших імен;
- текст макропідстановки – це фрагмент програми, яким буде замінено ім'я макроконстанти; він може містити імена макросів, визначених раніше, наприклад:

```
#define  PI  3.1415927
#define  PI3  (PI/3)
```

- директива `#define` виконує просту заміну вказаного макроса текстом підстановки; вираз заміни `(PI/3)` охоплено дужками, щоб операція ділення `PI` на 3 завжди виконувалась першою, зокрема вираз `x/PI3` буде замінено наступним:

```
x/(3.1415927/3)
```

- кожна директива препроцесора починається символом `#` і записується в окремому рядку; кінцем директиви є символ кінця рядка, знак `;` у директивах не ставиться.

Якщо в кінці виразу підстановки записати крапку з комою або інший знак, то цей знак буде сприйнято як останній символ тексту підстановки.

Функції програми. Наведена вище програма складається з двох функцій: функції `main()` та функції `Cels_to_Fahr()`. У функції `main()` реалізовано основні дії алгоритму поставленої задачі:

1) вводиться початкове значення температури в градусах Цельсія (рядки 13 і 14): функція `printf()` виводить на екран допоміжне повідомлення, а функція `scanf()` записує введене з клавіатури ціле число в змінну `tmin`;
2) виводиться заголовок (т. зв. "шапка") таблиці температур (рядки 15 і 16);
3) циклічно для кожного з `N` послідовних значень температури в градусах Цельсія обчислюється відповідне значення в градусах Фаренгейта і виводиться рядок таблиці результатів (рядки 17–21).

Формування таблиці виконано на основі оператора циклу `for`. Рядок 17 містить заголовок циклу, в якому задаються початкові значення змінних (`n=1` та `tCels=tmin`), умова виконання циклу (`n<=N`) та циклічна зміна параметра `n` (`n++` – збільшення на одиницю). Тіло циклу складається з трьох вкладених операторів (рядки 18–20). Для обчислення значення температури за Фаренгейтом, що відповідає заданому значенню `tCels`, у програмі використано окрему функцію `Cels_to_Fahr()`. Значення, яке повертає функція, присвоюється змінній `tFahr`. Функція `printf()` виконує виведення на екран двох значень температур: за Цельсієм (`tCels`) та за Фаренгейтом (`tFahr`). Специфікації `%10d` та `%21.1f` задають формати виведення для формування колонок таблиці. Наступний оператор (рядок 20) збільшує значення змінної `tCels` на заданий крок `DELTA_TC`.

Функція `Cels_to_Fahr()` складається з єдиного оператора `return`, який повертає в точку виклику даної функції значення виразу, що перетворює температуру із градусів Цельсія (параметр `tC`) у градуси Фаренгейта. Порядок запису функцій у

С-програмах довільний, зокрема в нашій програмі першою записано `main()`, а потім функцію `Cels_to_Fahr()`. Щоб компілятор міг проконтролювати правильність звертання до даної функції, в рядку 6 перед `main()` вказано т. зв. прототип функції – рядок заголовка, в якому зазначено ім'я функції, тип її значення та типи параметрів. Кожен прототип закінчується знаком `;`.

Як уже згадувалось, тіло функції складається з описів змінних і операторів. Змінні призначені для збереження даних програми. Фактично, *змінна* – це певна ділянка оперативної пам'яті, обсяг якої визначається типом, вказаним в описі змінної. У цю ділянку записуються поточні значення змінної. Описи змінних виконують на початку тіла функції перед першим оператором (рядки 9–12). Функція `main()` використовує чотири змінні: `tCels`, `tmin` та `n` оголошено як цілочислові змінні (тип `int`), а `tFahr` – як змінну дійсного типу (тип `double`). Функція `Cels_to_Fahr()` опрацьовує одну змінну `tC`, яка є формальним параметром даної функції. Додаткові внутрішні змінні у `Cels_to_Fahr()` не застосовуються.

2.3. Запис та оформлення програми

У мові С не встановлено конкретних правил щодо форми та стилю запису конструктивних компонентів програм – визначено тільки їх синтаксис. Проте для ефективної роботи з програмою дуже важливо, щоб форма її запису була наочною, послідовно однотипною, зручною для читання, легкою для сприйняття та коректування.

Запис операторів. С-програма складається з послідовності функцій. Тіло функції записується у фігурних дужках і формується з рядків описів та операторів. Ознакою кінця кожного опису чи оператора слугує знак `;`. Здебільшого всі прості оператори, а також вкладені оператори записують по одному в окремих рядках, як це зроблено в нашій демонстраційній програмі. Іноді доцільно записати в одному рядку два або три коротких оператори, чи розташувати довгий оператор на декількох рядках.

Заборонено розривати символами-роздільниками лексеми, що є ідентифікаторами, константами, ключовими словами чи складеними знаками операцій.

Конструктивні частини операторів та виразів доцільно виділяти, використовуючи для цього пробільні символи. Запис оператора

```
if (x>=xmin && x<=xmax)
    y=sin(x);
```

є істотно наочнішим, ніж поданий у наступному рядку:

```
if(x>=xmin&&x<=xmax)y=sin(x);
```

хоча згідно зі синтаксисом мови останній запис також правильний. Водночас зловживати роздільниками не слід, бо їх надмір погіршує читабельність програми. Запис

```
c=(a+b)/2;          /* середина інтервалу пошуку */
```

буде більш зрозумілим, ніж наступний, який синтаксично і семантично повністю еквівалентний до попереднього:

```
c = (a    /*    ліва межа інтервалу пошуку    */    +
b    /*    права межа інтервалу пошуку    */    ) / 2  ;
```

Щоб виділити тіло функції, описи та оператори звичайно починають, відступивши на декілька позицій від лівого краю програми – "втоплюють" відносно фігурних дужок, якими охоплено тіло функції. Початки операторів одного рівня (тобто тих, що виконуються послідовно) прийнято вирівнювати по вертикалі (рядки 13-15, 17, 22, а також 18-20). Оператори, які є складовими частинами інших операторів, зокрема ті, що формують тіло циклу, відсувають відносно початку оператора заголовка циклу (рядки 18-20). Так само "втоплюють" внутрішні оператори умовних операторів (`if` та `switch`). Відступи в записах вкладених операторів підкреслюють структуру програми та підвищують її наочність, тому нехтувати ними не можна. Оптимальним розміром відступу для вкладених операторів є 2-4 позиції.

Розміщення фігурних дужок. Фігурні дужки `{` і `}`, якими відповідно починається і завершується тіло кожної функції, як правило, записують в окремих рядках, вирівнюючи їх по вертикалі (рядки 8, 23 і 26, 28 нашої програми).

Фігурними дужками в мові С також виділяють *блок* – послідовність операторів, яка синтаксично розглядається як один оператор. Блоки використовують для вкладення групи операторів у оператори циклу та в умовні оператори. Найбільш поширеними серед програмістів є два способи запису блоків, які проілюструємо наступними прикладами. Перший спосіб – виділення блоку відступами та окремими рядками:

```
while ( numb > 0 )
  {
     sum = sum + numb % base;
     numb = numb / base;
  }
```

У другому популярному способі запису блоків застосовують тільки один відступ:

```
while ( numb > 0 ) {
   sum = sum + numb % base;
   numb = numb / base;
}
```

З наведених прикладів видно, що перший спосіб запису блоків виразніше виділяє блок, але використовує додатковий рядок. Крім цього, у разі великої кількості вкладень окремі відступи для дужок і тіла блоку призводять до значних зсувів рядків управо, що погіршує читабельність тексту програми. У наведеній вище демонстраційній програмі використано другий спосіб запису блоків: дужка початку блоку `{` записується в рядку заголовка циклу чи умовного оператора, а дужка кінця блоку `}` – в окремому рядку в тій самій позиції, з якої починався заголовок циклу (рядки 17 і 21) чи інший складений оператор. Такого стилю запису будемо дотримуватись і в усіх наступних програмах.

Коментарі. Важливу роль у підвищенні наочності та зрозумілості програм відіграють коментарі, хоча вони не є обов'язковими компонентами програми і розгляда-

ються компілятором як звичайні роздільники лексем – тексти коментарів вилучаються з програми під час препроцесування. Коментар починається парою символів /*, а закінчується зворотною парою – */. Коментар може займати цілий рядок програми (рядок 24), декілька рядків (рядки 1,2) або частину рядка (рядки 9-12). Мова С не дозволяє вкладати один коментар у інший, тому якщо закоментувати цілий фрагмент програми, в якому вже містяться коментарі, то компілятор зафіксує помилку.

Приклад помилкового вкладення коментарів:

```
/*
    d=5*x;
    h=d*d+0.6*x;        /* граничне значення */
    c=(d+h)/2;
*/
```

Стандарт С-99 дозволив застосовувати ще один вид коментарів – однорядкові коментарі, які широко використовуються мовою С++ і стали дуже популярними. Однорядковий коментар починається двома косими рисками (правими слешами) //, а завершується символом нового рядка:

```
. . .                   // це С++ і С-99 коментар
```

Надалі у програмах для універсальності та наочності будемо використовувати тільки класичні С-коментарі: /* це коментар С */. Проте, якщо в системі програмування, з якою працює читач, підтримуються однорядкові коментарі, то рекомендуємо застосовувати цей вид коментарів у програмах (зокрема, в разі створення програм у середовищі Borland C++). Однорядкові коментарі є не лише коротшими і легшими для запису, але й можуть бути вкладеними у зовнішній С-коментар. Остання властивість широко використовується під час налагодження програм. Наступний закоментований фрагмент програми буде сприйнятий компілятором як синтаксично правильний коментар:

```
/*
    d=5*x;
    h=d*d + 0.6*x;      // граничне значення
    c=(d+h)/2;
*/
```

2.4. Етапи виконання програми

Безпосередньому виконанню програми, написаної мовою С, передує декілька етапів перетворення її програмного коду, основними з яких є: введення тексту програми, препроцесорне опрацювання та компіляція програмних елементів, компонування об'єктних кодів складових частин програми в єдиний виконавчий код (рис. 2.1). Розглянемо коротко кожен з цих етапів.

Введення програми. Майже всі системи програмування мають вбудовані текстові редактори, призначені для швидкого введення і зручного редагування програм. Наприклад, редактор текстів інтегрованого середовища Borland C візуалізує тексти програм,

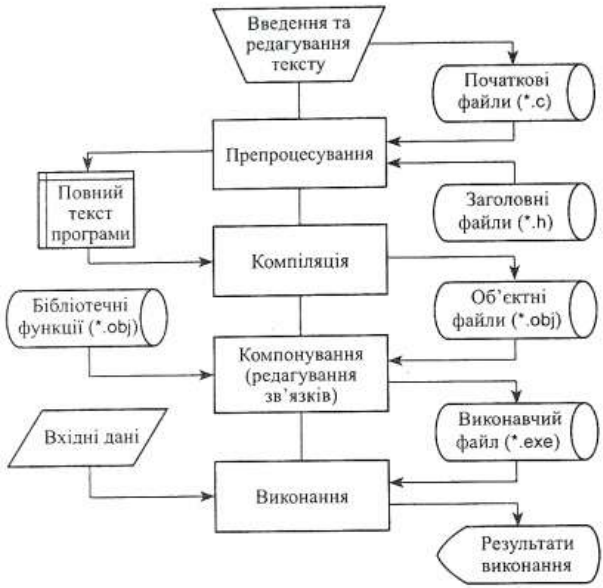


Рис. 2.1. Схема виконання С-програм

виділяючи кольорами їх основні компоненти: директиви препроцесора, службові слова, константи, ідентифікатори, коментарі; спрощує процес вертикального вирівнювання та форматування рядків програми; підтримує основні засоби редагування: копіювання і перенесення блоків тексту, пошук та заміну, швидке пересування по програмі тощо. Редактор текстів інтегрованого середовища пов'язаний з компілятором та засобами налагодження програм, що дає змогу встановлювати місця синтаксичних помилок у тексті програми та організовувати контроль за ходом виконання програми.

Результатом етапу введення програми є т. зв. *початковий код* – один або декілька файлів із текстами складових частин програми. Прийнято, що файли з текстами С-програм мають розширення .c, а файли з програмами, написаними мовою C++, дістають розширення .cpp.

Препроцесування та компіляція програми. Як вже зазначалось, до складу С-програм входять директиви препроцесора – рядки, що починаються знаком #. Спеціальна програма, яку називають препроцесором, опрацьовує програму перед компіляцією, послідовно виконуючи всі задані директиви, зокрема доповнює текст програми заголовними файлами, вказаними у директивах `#include`, виконує макрозаміни та макропідстановки, задані директивами `#define`, вилучає всі коментарі тощо. Результатом роботи препроцесора є повний текст програми, призначений для компіляції.

Стандартно в середовищі програмування Borland C (як і в багатьох інших системах програмування) текст програми після препроцесування не записується в спеціальний файл, а

відразу передається для опрацювання компілятору. Якщо ж потрібно переглянути результат роботи препроцесора, то слід автономно запустити програму-препроцесор (детальніше про це мова йтиме в розділі 17).

Наступним етапом опрацювання програми є компіляція – переведення повного текстового коду програми в систему машинних (тобто зрозумілих для комп'ютера) інструкцій. Таке перетворення виконує окрема програма, яка називається компілятором (*Compiler*). Компілятор перевіряє синтаксичну правильність програми і в разі наявності помилок формує список відповідних повідомлень. Це означає, що потрібно повернутись до етапу редагування програми, проаналізувати та виправити всі виявлені помилки.

Якщо програма синтаксично правильна, то відбувається процес компілювання, результатом якого є файл з т. зв. *об'єктним кодом* програми – набором двійкових інструкцій (*машинних команд*), які можуть бути виконані комп'ютером. Файли об'єктних кодів мають, як правило, такі ж імена як і файли початкових кодів та розширення .obj. Якщо текст нашої першої програми записати в файл myfirst.c, то в результаті компіляції буде створено файл об'єктного коду програми myfirst.obj.

Компонування виконавчого коду. Переважна більшість програм мовою С здійснює звертання до стандартних бібліотечних функцій, оскільки такі операції, як введення чи виведення даних, обчислення математичних функцій, робота з файлами, збереження даних у динамічній пам'яті та низка інших не входять до вбудованих засобів компілятора – вони реалізовані як окремі функції. Об'єктні коди цих функцій зберігаються у спеціальних *бібліотеках*, які згідно зі стандартом обов'язково входять до складу кожної системи програмування.

Користувач може також розробити і відкомпілювати власні функції, а потім занести їх об'єктні коди в бібліотеку. Надалі такі функції можна буде використовувати так само, як і стандартні бібліотечні функції та бібліотечні функції даної системи програмування.

На етапі компонування програма-компонувальник (її також називають редактором зв'язків – *Link Editor*) під'єднує до об'єктного коду програми коди тих бібліотечних функцій, звертання до яких виконуються в даній програмі. У результаті створюється готовий до виконання код програми, який заноситься у файл з розширенням .exe – *виконавчий код* програми. Зокрема, після опрацювання компонувальником об'єктного коду з файла myfirst.obj буде створено файл myfirst.exe, який є виконавчим файлом програми myfirst.c.

Коли розробляються великі програми чи проекти, то найчастіше їх розділяють на декілька програмних файлів, кожен з яких створюють і компілюють автономно. У цьому випадку редактор зв'язків об'єднує об'єктні коди окремих програмних файлів та використаних бібліотечних функцій у єдиний exe-код програми.

Виконання exe-коду програми. Файли з розширенням .exe містять виконавчі коди програм – їх можна запустити на виконання зі середовища програмування чи безпосередньо з операційного середовища. Процес виконання програми здебільшого вимагає введення певних початкових даних, які можуть надходити з клавіатури, зчитуватись з дискового файла чи передаватись із якогось зовнішнього пристрою. Результати

реалізації програми найчастіше виводяться на екран дисплея, можуть бути роздруковані на паперовому носії, записані в файл чи передані в канал зв'язку тощо.

Готові програми слід обов'язково протестувати, щоб переконатися, що вони коректно працюють для різних наборів вхідних даних, зокрема, відстежують введення помилкових даних. Якщо виявлено хибність результатів виконання програми, то слід внести необхідні виправлення у початковий код програми та повторити всі наступні етапи реалізації.

Запитання та завдання для самоконтролю

1. З чого складається С-програма? Які види функцій використовуються у програмах?
2. У чому особливість функції `main()`?
3. З чого формуються функції? Як записують функції у С-програмах?
4. Що таке заголовний файл? Як під'єднується заголовний файл до програми?
5. Яке призначення наступної директиви препроцесора:
   `#define MAX 250`
6. Як записують оператори у С-програмах? З якою метою застосовують відступи у записах операторів? Які способи відступів і вирівнювання найбільш наочні?
7. В яких місцях програми можна записувати коментарі? Який синтаксис коментарів?
8. Що є вхідними даними і результатом кожного з етапів реалізації програм: введення і редагування, препроцесування, компіляції, компонування, виконання?
9. Проаналізуйте, що буде виконувати наступна програма:

```
#include <stdio.h>
void FrameLine (int k);      /* прототип функції */
int main (void)
{
   FrameLine(10);
   printf ("\t*  Вивчаємо мову С  *\n");
   FrameLine(10);
   return 0;
}
void FrameLine (int k)
{
     int i;
   printf ("\n\t");
   for (i=1; i<=k; i++)
     printf ("*=");
   printf ("*\n");
}
```

10. Введіть текст наведеної вище програми, відкомпілюйте та виконайте її. Перевірте результат. Потім поміняйте місцями функції `main()` та `FrameLine()`. Прототип функції `FrameLine()` закоментуйте. Повторно відкомпілюйте та виконайте програму. Чи змінився результат?

Розділ 3

ТИПИ ДАНИХ

У цьому розділі:

- Класифікація типів даних
- Характеристики базових цілочислових типів `int` та `char`, а також типів, утворених за допомогою модифікаторів `signed`, `unsigned`, `short` і `long`; цілочислові та символьні константи; макроконстанти граничних значень
- Дійсні типи даних: `float`, `double` та `long double`, збереження в пам'яті та параметри; дійсні константи з фіксованою крапкою і з плаваючою крапкою
- Описи та ініціалізація змінних, глобальні та локальні змінні
- Переліки: оголошення, властивості, застосування; перелікові змінні, відображення значень перелікових змінних

Кожне дане програми (змінна чи константа) має певний тип. *Тип* задає обсяг оперативної пам'яті, яка виділяється для збереження цього даного, визначає діапазон допустимих значень даного та встановлює операції, які можуть виконуватись з цим даним. Тип змінних у С-програмах завжди вказується явно в їх описах, а тип констант встановлюється компілятором автоматично за формою запису константи в програмі.

3.1. Класифікація типів даних

Всі типи мови С можна поділити на три групи:

- *скалярні* (або *прості*) типи, які в довільний момент часу можуть мати тільки одне значення; скалярними є арифметичні типи, переліки та вказівники;
- *агреговані* (або *складені*) типи, які формуються за встановленими правилами з наборів скалярних типів; до агрегованих типів належать: масиви, структури та об'єднання;
- тип "функція", що оголошує функції зі заданим складом параметрів і встановленим типом значення, яке повертає функція.

Арифметичні типи можна поділити ще на дві групи:

- *цілочислові* типи, дані яких можуть набувати тільки цілих значень (знакових або беззнакових) і зберігаються в форматах, встановлених для цілих чисел;
- *дійсні* типи, дані яких є дійсними числами і зберігаються в оперативній пам'яті в форматах чисел з плаваючою крапкою.

Крім цього, типи даних поділяють на *базові* (інші назви: *основні*, *фундаментальні*) та *похідні*. Базовими вважаються типи: `char`, `int`, `float`, `double`, а також типи, утворені з них через застосування *модифікаторів*: `short`, `long`, `signed`, `unsigned` (їх також називають *кваліфікаторами* типів). Особливим базовим типом є `void` – *порожній* або *невизначений* тип. Тип `void`, зокрема, застосовують в оголошеннях функцій, щоб зазначити, що функція не повертає ніякого значення, або щоб вказати, що функція не використовує параметрів.

Похідними вважаються вказівникові типи, агреговані типи та функції. Всі вони формуються з використанням базових типів.

У цьому розділі детально розглянемо лише базові типи мови С та переліки, похідні типи будуть розглянуті далі.

3.2. Цілочислові типи

Базовими цілочисловими (або просто *цілими*) типами є `int` та `char`.

Тип `int`. Основним типом для роботи з цілочисловими даними є тип `int` (від слова *integer* – ціле число). Згідно зі стандартом мови С розмір типу `int` відповідає розміру машинного слова, властивому програмно-апаратній платформі конкретної системи програмування. Для переважної більшості сучасних комп'ютерів розмір `int` становить 2 або 4 байти, зокрема в середовищі Borland C дані з типом `int` мають розмір 2 байти (16 бітів).

До типу `int` можна застосувати модифікатори `long` (довге ціле) та `short` (коротке ціле). Модифікатор типу `long` вказує, що дане буде займати не менше як 4 байти (у більшості систем програмування тип `long int` має розмір саме 4 байти). Модифікатор `short` вказує, що дане не повинно перевищувати розмір типу `int` (у більшості реалізацій тип `short int` займає 2 байти). Діапазони допустимих значень цілих типів, встановлені у Borland C, наведено в табл. 3.1.

Тип `int` та його модифікації є знаковими типами. Це означає, що один біт (здебільшого найстарший) двійкового коду числа займає знак. Прийнято, що знак плюс позначається нулем, а знак мінус – одиницею. Від'ємні цілі числа у більшості комп'ютерних реалізацій зберігаються в т. зв. *доповнювальному* коді.

Доповнювальний код формується шляхом інвертування всіх бітів числа (крім знакового) і наступного додавання одиниці до отриманого зворотного коду. Застосування доповнювального коду дає змогу уникнути апаратного виконання операції віднімання – замість віднімання виконується додавання доповнювального коду числа.

Якщо певні дані набувають тільки додатних значень, то можна вдвічі збільшити діапазон їх додатних значень, оголосивши такі дані з модифікатором `unsigned` (беззнаковий). У беззнакових даних біт знака розглядається як звичайний числовий біт. Стандарт С також підтримує модифікатор `signed`, який підкреслює, що дане є знаковим, проте в оголошеннях змінних записи `signed int` використовують рідко, оскільки тип `int` завжди є знаковим. Обидва модифікатори `signed` та `unsigned` можна застосовувати і до даних з типами `long int` або `short int`.

Таблиця 3.1

Параметри цілочислових типів у середовищі Borland C

Тип	Розмір (у байтах)	Діапазон значень
`signed char`	1	від -128 до 127
`unsigned char`	1	від 0 до 255
`int`	2	від -32768 до 32767
`unsigned int`	2	від 0 до 65535
`short int`	2	від -32768 до 32767
`unsigned short int`	2	від 0 до 65535
`long int`	4	від -2147483648 до 2147483647
`unsigned long int`	4	від 0 до 4294967295
`long long int`*	8	від -2^{63} до $2^{63}-1$
`unsigned long long int`*	8	від 0 до $2^{64}-1$

* тільки для систем програмування, що реалізують стандарт С-99

Стандарт С-99 ввів ще один цілий тип – `long long int` (дуже довге ціле), призначений для комп'ютерів, які підтримують 64-розрядні (8-байтові) цілі числа. До цього типу теж можна застосовувати модифікатор `unsigned` (див. табл. 3.1).

Цілочислові константи. Нагадаємо, що цілочислові константи можна записувати в трьох формах: десятковій, вісімковій та шістнадцятковій. *Десяткова* константа складається з послідовності десяткових цифр, перед якими може стояти знак `+` або `-`. *Вісімкова* константа завжди починається цифрою `0`, за якою записуються вісімкові цифри (від `0` до `7`), що формують значення константи. *Шістнадцяткова* константа починається префіксом `0x` або `0X` (нуль і маленька чи велика літера `x`), за яким вказуються шістнадцяткові цифри значення константи (цифри `0`..`9` та літери `a`..`f` або `A`..`F`). Вісімкові та шістнадцяткові константи теж можуть записуватись зі знаком, але здебільшого їх використовують як беззнакові числа. Приклади:

```
147    +200795    -63            – десяткові константи;
036    0147    03067723          – вісімкові константи;
0xa5f0    0x147    0XFFFF0000    – шістнадцяткові константи.
```

Звернемо увагу, що три записані вище константи: `147`, `0147` та `0x147` мають різні значення та різні двійкові коди. Десяткове значення константи `0147` дорівнює 103, а константи `0x147` – 327.

Оскільки в мові С не підтримуються двійкові константи, то для роботи з двійковими кодами чисел найчастіше використовують їх вісімкові або шістнадцяткові еквіваленти. Перехід від двійкового коду числа до його вісімкового чи шістнадцяткового зображення є простим і ґрунтується на тому, що кожна вісімкова цифра відповідає двійковій тріаді (тобто трьом послідовним бітам: $8 = 2^3$), а кожна шістнадцяткова цифра – двійковій тетраді (чотирьом послідовним бітам: $16 = 2^4$). Щоб записати вісімкове значення двійкового числа, треба поділити

його на тріади, починаючи від наймолодшого розряду, і замінити кожну тріаду відповідною вісімковою цифрою. Аналогічно, щоб отримати шістнадцяткове значення двійкового числа, треба розбити це число на тетради і замінити тетради шістнадцятковими цифрами. Наприклад, щоб написати вісімкову константу, рівну двійковому числу 100111011000010, розіб'ємо це число на тріади: 100 111 011 000 010 і запишемо кожну тріаду вісімковою цифрою. Результат буде таким: `047302`. Щоб отримати шістнадцяткову константу, значення якої дорівнює цьому двійковому числу, розбиваємо число на тетради: 0100 1110 1100 0010 і записуємо кожну тетраду шістнадцятковою цифрою: `0x4ec2`. Зворотні перетворення – з вісімкової та шістнадцяткової систем у двійкову – виконують так само: кожну вісімкову цифру записують еквівалентною двійковою тріадою, а кожну шістнадцяткову – двійковою тетрадою.

Тип цілочислової константи встановлюється компілятором за її значенням і формою запису. При цьому застосовуються такі правила:

- якщо значення константи потрапляє в діапазон значень типу `int`, то така константа отримує тип `int`;
- інакше, якщо це беззнакова вісімкова чи шістнадцяткова константа, яка потрапляє в діапазон значень типу `unsigned int`, то їй присвоюється тип `unsigned int`;
- інакше, якщо значення константи потрапляє в діапазон значень типу `long int`, то ця константа отримує тип `long int`;
- інакше, якщо константа беззнакова і потрапляє в діапазон `unsigned long int`, то їй присвоюється цей тип;
- інакше фіксується помилка.

Зокрема, записані вище константи: `147`, `-63`, `036`, `0147` та `0x147` отримають тип `int`, константа `0xa5f0` – тип `unsigned int`, константи `+200795` та `03067723` отримають тип `long int`, а константа `0XFFFF0000` – тип `unsigned long int`.

Мова С дає змогу модифікувати тип константи. Для цього до чиєла долучають кінцеві символи `u` (або `U`) та `l` (або `L`). Символ `u/U` вказує, що константі має бути присвоєний модифікатор `unsigned`, а символ `l/L` розширює константу до розміру типу `long int`. Можна поєднувати обидва символи модифікації типу в довільному порядку. Подамо три приклади модифікованих констант:

`47365u` – константа отримає тип `unsigned int`;
`07773L` – константа отримає тип `long int`;
`0x1bd72ul` – константа отримає тип `unsigned long int`.

Символьні константи. До цілочислових констант належать також символьні константи. *Символьна константа* – це довільний, допустимий для встановленої на комп'ютері кодової таблиці, графічний символ, керуючий слеш-символ або ескейп-послідовність, записані в апострофах.

Приклади символьних констант:

`'F'` `'n'` `'+'` `'7'` `'й'` `'Ж'` `'|'` – константи графічних символів;
`'\n'` `'\t'` `'\b'` `'\a'` – константи керуючих послідовностей;
`'\\'` `'\"'` `'\0'` `'\310'` `'\x1b'` – константи ескейп-послідовностей.

Стандартно символьні константи зберігаються в форматі `int`, хоча у системах програмування, які застосовують ASCII-коди або іншу однобайтову систему кодуван-

ня, старший байт надлишковий. Значенням символьної константи є числове значення коду символа. Саме код символа бере участь у всіх операціях, в які вступає символьна константа. Тому одну і ту ж саму за значенням символьну константу можна записати різними способами, наприклад: `'+'` – символьне зображення, `'\052'` – вісімкова ескейп-послідовність, `'\X2A'` – шістнадцяткова ескейп-послідовність. Можна також скористатись значенням коду і записати символ `'+'` як десяткове число `42`.

Звернемо увагу, що дві наступні символьні константи `'0'` та `'\0'` є різними за значенням. Перша є символом цифри нуль (її ASCII-код 48), а друга позначає т. зв. нуль-символ (її ASCII-код 0). Так само різними є константи `'a'`, `'\a'` та `'\xa'`. Перша позначає відповідну латинську літеру (ASCII-код 97), друга – символ стандартного звукового сигналу (керуюча послідовність, що відповідає символу з ASCII-кодом 7), а третя – символ нового рядка (ASCII-код 10).

Використання символьних констант (зокрема констант керуючих послідовностей) замість ASCII-кодів символів поліпшує читабельність програми і робить її мобільною, тобто незалежною від системи кодування символів.

Тип char. Мова С відносить тип `char` (від слова *character* – літера) до цілочислових типів. У переважній більшості реалізацій С дані з типом `char` займають один байт оперативної пам'яті (хоча є системи, в яких тип `char` двобайтовий) і розглядаються як знакові цілі числа. До типу `char` можна застосовувати модифікатори `signed` та `unsigned`. Модифікатор `signed` явно встановлює знаковість даного з типом `char`, а модифікатор `unsigned` робить його беззнаковим (див. табл. 3.1).

В інтегрованому середовищі Borland C знаковість/беззнаковість типу `char` можна встановити через відповідну опцію середовища (пункт меню Options/Compile/Code generation/Unsigned chars). Все ж, якщо знак даних з типом `char` може вплинути на результати роботи програми, то доцільніше явно вказувати модифікатор `signed` або `unsigned` в оголошеннях `char`-змінних.

Оскільки тип `char` належить до цілих типів, то він сумісний з усіма арифметичними типами. Все ж основним застосуванням типу `char` є збереження кодів символів у символьних рядках.

Зауважимо також, що в оголошеннях змінних, значеннями яких будуть одиночні символи, у С-програмах переважно використовують тип `int`, а не `char`. Це дає змогу уникнути перетворення типів, пов'язаного з тим, що більшість операцій розширює тип `char` до розміру `int` (доповнює значення символа старшим нульовим байтом). Саме тому символьні константи мови С теж зберігаються у форматі `int`.

Широкі символи. Для підтримки національних мов, що мають багатосимвольні алфавіти, спеціальним доповненням до стандарту С у 1995 р. введено тип `wchar_t`, призначений для двобайтового кодування символів. Цей тип не належить до вбудованих типів мови С, для його використання треба підключити заголовний файл `<stddef.h>`. Символи з типом `wchar_t` називають *широкими* (або *розширеними* – *wide-characters*). Константи широких символів починаються літерою `L`, наприклад: `L'A'`, `L'2'`, `L'$'`.

Стандарт С-99 затвердив великий набір бібліотечних функцій, які реалізують операції з широкими символами (див. Додаток 3).

У середовищі Borland C тип `wchar_t` не має практичного застосування, оскільки він задекларований як тотожний до базового типу `char`.

Макроконстанти граничних значень. В заголовному файлі `<limits.h>`, що належить до стандартних заголовних файлів мови С, записано набір макроконстант (іменованих констант), значення яких дорівнюють мінімальним і максимальним значенням, встановленим для цілих типів у даній системі програмування. Наведемо основні макроконстанти граничних цілочислових значень:

`CHAR_MIN, CHAR_MAX` – найменше і найбільше значення даних з типом `char`;
`SCHAR_MIN, SCHAR_MAX` – діапазон значень даних з типом `signed char`;
`INT_MIN, INT_MAX` – діапазон значень даних з типом `int`;
`SHRT_MIN, SHRT_MAX` – діапазон значень даних з типом `short int`;
`LONG_MIN, LONG_MAX` – діапазон значень даних з типом `long int`;
`LLONG_MAX` – найбільше значення даних з типом `long long int`.

Для беззнакових цілих типів визначено тільки макроконстанти верхніх граничних значень: `UCHAR_MAX`, `UINT_MAX`, `USHRT_MAX`, `ULONG_MAX` та `ULLONG_MAX`. Мінімальні значення усіх беззнакових типів дорівнюють 0. Макроконстанта `CHAR_BIT` з набору `<limits.h>` задає розмір типу `char` у бітах.

Названі макроконстанти доцільно використовувати в програмах, щоб, по-перше, унаочнити записи і уникнути довгих числових констант, а по-друге, щоб зробити програму мобільною, тобто незалежною від апаратно-програмних характеристик комп'ютера, на якому компілюється чи виконується програма. Нагадаємо, що для під'єднання заголовного файла треба застосувати директиву

```
#include <limits.h>.
```

3.3. Дійсні типи

Для збереження й опрацювання дійсних чисел (тобто чисел, що складаються з цілої та дробової частин) мова С підтримує три дійсних типи: `float`, `double` і `long double`. Стандарт С не задає внутрішніх форм збереження даних для цих типів і точних діапазонів їх значень – вони визначаються конкретними реалізаціями, встановлено тільки, що тип `float` повинен мати точність не менше 6-ти значущих десяткових цифр, а типи `double` і `long double` – не менше 10-ти значущих десяткових цифр. Діапазон значень всіх трьох дійсних типів згідно зі стандартом повинен бути не меншим, ніж від 10^{-37} до 10^{37}.

У Borland C для збереження даних усіх дійсних типів застосовується формат чисел з плаваючою крапкою (рис. 3.1). Такий формат встановлює, що число (X) складається з двох частин: мантиси (m) і порядку (p):

$$X = m \cdot 2^p, \quad 0{,}1 \le m < 1.$$

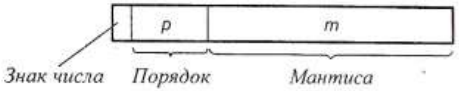


Рис. 3.1. Формат збереження даних дійсного типу

Старший біт числа займає знак мантиси, за ним записується двійковий порядок числа, а потім – сама мантиса. Мантиса зберігається в нормалізованій формі, тобто старшою цифрою мантиси завжди є двійкова одиниця (нормалізація мантис виконується шляхом зсуву їх уліво з відповідним зменшенням значення порядку). Розміри мантиси і порядку для трьох дійсних типів є різними (табл. 3.2), вони визначають точність чисел і діапазон їх значень.

Таблиця 3.2

Параметри дійсних типів у середовищі Borland C

Тип	Розміри			Точність мантиси (десятк. цифр)	Діапазон десяткового порядку
	загальний (байти)	*m* (біти)	*p* (біти)		
`float`	4	23	8	7	від -38 до 38
`double`	8	52	11	16	від -308 до 308
`long double`	10	64	15	20	від -4932 до 4932

Зокрема, дані з типом `double` займають в оперативній пам'яті 8 байтів, з них 53 біти відведено для мантиси, решту – для порядку. Такий розподіл дає змогу зберігати дійсні числа з точністю, що відповідає приблизно 16-ти десятковим цифрам. Найменше додатне число цього типу має значення $2{,}225 \cdot 10^{-308}$, а найбільше – $1{,}798 \cdot 10^{308}$.

Константи дійсних типів. Для запису дійсних констант використовують дві форми: з *фіксованою крапкою* та з *плаваючою крапкою*. В записах чисел з фіксованою крапкою дробова частина числа відокремлюється від цілої десятковою крапкою. В записах чисел з плаваючою крапкою вказуються десяткові значення мантиси і порядку, розділені літерою `e` або `E`. Мантиса може записуватись як ціле число або як дійсне число з фіксованою крапкою; порядок завжди цілий (зі знаком або беззнаковий). Першим символом числа вказують знак мантиси (якщо він потрібний). Наприклад:

```
17.168    -0.035    +1843.6   – константи з фіксованою крапкою;
0.14352e-4    -17.35E8    297e+12   – константи з плаваючою крапкою.
```

Незалежно від форми запису дійсна константа зберігається в пам'яті як число з плаваючою крапкою в форматі `double`.

Тип дійсної константи можна модифікувати, долучивши до неї кінцеву літеру `f` (або `F`) для встановлення типу `float` чи літеру `l` (або `L`) – для типу `long double`. Приміром, константи `78.256f` та `0.45E-3F` будуть зберігатись у форматі `float`, а константи `8.5564L` та `-11.65e+451` – у форматі `long double`.

У заголовному файлі `<float.h>` записані макроконстанти, які характеризують параметри всіх трьох дійсних типів (`float`, `double` і `long double`), зокрема: розміри даних, найменші та найбільші допустимі значення, точність мантиси (у бітах і в

десяткових цифрах), діапазони значень порядку для основи 2 і для основи 10 тощо. Використання цих макроконстант є особливо корисним у разі, якщо програма буде налагоджуватись і реалізовуватись у різних системах програмування на комп'ютерах різних типів.

3.4. Описи змінних

Всі змінні С-програми як внутрішні, так і глобальні повинні бути описані явно (оголошені). Синтаксис оголошення змінних такий:

тип список_змінних;

тут *тип* – ім'я одного з базових або похідних типів з можливими модифікаторами та кваліфікаторами; *список_змінних* – послідовність з одного або декількох ідентифікаторів, відокремлених комами, що задають імена змінних даного типу.

Подамо кілька прикладів оголошення змінних:

```
int  i, k, letter;
unsigned int  n1, n2, nstep;
long int  sum;
double  result;
```

В оголошеннях даних, що мають цілочисловий тип, утворений на основі типу `int` через модифікатори `signed/unsigned` та `short/long`, ключове слово `int` можна опускати і записувати тільки слова-модифікатори.

Другий і третій рядок попередніх оголошень можна записати так:

```
unsigned  n1, n2, nstep;
long  sum;
```

Ініціалізація змінних. Описуючи змінну, можна відразу надавати їй початкове значення – *ініціалізувати* дану змінну. Для ініціалізації після імені змінної записують знак присвоєння `=`, а за ним вираз, значення якого отримає ця змінна. Елементами виразу ініціалізації можуть бути константи або змінні, значення яких вже відомі. Приклади описів з ініціалізацією:

```
int  m=10, symb='*';
double  eps=0.5e-6;
unsigned  kr=5, next=m+kr;
```

Змінна, в описі якої перед типом вказано кваліфікатор `const`, вважається сталою (константною) – змінювати її ініціалізаційне значення заборонено. Наприклад, константу 2π можна оголосити так:

```
const double  TwoPi=2*3.1415926;
```

Раніше говорилося, що для роботи з константними значеннями в мові С широко застосовують макропідстановки (макроконстанти), які реалізуються через директиву препроцесора `#define`. Використання константних змінних є альтернативним способом

програмування констант. Перевага цього способу в тому, що дані, описані з кваліфікатором `const`, зберігають усі властивості змінних (зокрема, мають визначений тип і встановлену область дії) – заборонено тільки змінювати значення, яких вони набули після ініціалізації. Ще одна важлива перевага константних змінних – вони дають змогу уникнути побічних ефектів, які можуть виникнути, коли виконується макропідстановка виразу (див. параграф 17.2).

Якщо в описі змінної вказано кваліфікатор `volatile`, то така змінна може отримувати значення від зовнішнього середовища (наприклад, від операційної системи або зовнішньої програми) незалежно від програми, в якій вона використовується. Зазначені змінні найчастіше застосовують для зв'язку з апаратними пристроями комп'ютера: портами, таймером тощо.

Глобальні та локальні змінні. Область дії змінної залежить від місця програми, в якому описано (оголошено) дану змінну. Оголошення змінних можна виконувати:

- між функціями – такі змінні називаються *глобальними*, вони можуть використовуватись у всій програмі, починаючи від точки їх оголошення; глобальні змінні автоматично ініціалізуються нульовим значенням;
- всередині функцій або програмних блоків – такі змінні називаються *локальними*; область дії локальної змінної обмежена функцією чи програмним блоком, в якому оголошено дану змінну;
- в списку формальних параметрів функцій – такі змінні теж локальні та діють у межах функції або її прототипу.

Переважна більшість змінних кожної програми є локальними. Локальні змінні існують тільки під час виконання функції чи програмного блоку, в яких вони оголошені. Ці змінні можна використовувати від точки їх оголошення до кінця даної функції або програмного блоку. Треба пам'ятати, що локальні змінні не ініціалізуються автоматично. Тому, якщо при описі локальній змінній не надано початкового значення, то в ділянці, яку займає ця змінна, буде т. зв. "сміття", тобто те, що було записане в цю ділянку раніше. Детально питання видимості та часу існування глобальних і локальних змінних розглядаються в розділі 12.

За стандартом С-89 всі оголошення локальних змінних повинні записуватись на початку функції або відповідного програмного блоку перед першим оператором. Мова С++, а потім і стандарт С-99 дозволили оголошувати змінні всередині операторної частини функцій перед першим звертанням до змінних. Проте практика засвідчує, що програма є наочнішою та відчутно зручнішою для редагування і доповнення, якщо описи змінних згруповані та розміщені у визначених місцях програми.

3.5. Переліки

Перелікові типи (їх здебільшого називають просто *переліки*) – це набори цілочислових констант, кожна з яких має унікальне ім'я. Оголошують переліки так:

```
enum тег_переліку {список_іменованих_констант};
```

тут *тег_переліку* – ім'я, яке визначає (ідентифікує) даний перелік; *список_іменованих_констант* – набір імен, які надаються константам переліку. Тег переліку та імена його констант повинні відповідати правилам запису ідентифікаторів. Один з усіх переліків програми можна не іменувати тегом, а тільки відзначити ключовим словом `enum`.

Наступний перелік надає імена константам, які позначають дні тижня:

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

Кожен елемент переліку є константою з типом `int`, йому присвоюється значення, на одиницю більше, ніж значення попереднього елемента. Стандартно перший елемент переліку отримує значення нуль. Тобто константа `Mon` з переліку `days` дорівнює 0, константа `Tue` – 1, константа `Wed` – 2 і т.д. Стандартні значення констант можна змінити, присвоївши їм потрібні цілочислові значення:

```
enum work_days {MON=1, TUE, WED, THU, FRI};
```

У переліку `work_days` перший елемент `MON` отримає значення 1, а кожен наступний – значення, більше на 1. Приміром, константа `FRI` дорівнюватиме 5.

Константне значення можна присвоювати всім або лише декільком елементам переліку:

```
enum RGBcolors {Blue=1, Green, Red=4};
```

У даному переліку константа `Blue` отримає значення 1, `Green` – 2, а константа `Red` – 4. Якби далі в списку була ще одна іменована константа, то вона отримала б значення, наступне за `Red`, тобто 5.

Оскільки константи переліків мають тип `int`, то вони сумісні з усіма арифметичними типами мови С.

Переліки є зручним способом для створення груп іменованих констант. У свою чергу, застосування іменованих констант значно підвищує наочність програми, спрощує внесення змін і доповнень, полегшує налагодження. У заголовних файлах `<conio.h>` та `<graphics.h>` бібліотеки Borland C оголошено ряд переліків, що широко використовуються у процесах формування текстових і графічних екранних зображень. Ці переліки складаються з констант, які іменують набори допустимих значень кольорів, стилі елементів зображення, номери векторних шрифтів, режими роботи відеосистеми тощо. Зокрема, іменовані константи, що задають 16-елементну палітру кольорів, оголошено так:

```
enum COLORS {
     BLACK,              /* темні кольори */
     BLUE,  GREEN,  CYAN,  RED,  MAGENTA,  BROWN,  LIGHTGRAY,
     DARKGRAY,           /* світлі кольори */
     LIGHTBLUE,  LIGHTGREEN,  LIGHTCYAN,  LIGHTRED,
     LIGHTMAGENTA,  YELLOW,  WHITE
};
```

Тобто, чорний колір можна задавати значенням 0 або імснованою константою BLACK, синій – значенням 1 або іменованою константою BLUE і т. д. Подамо два приклади звертання до бібліотечних функцій з <conio.h> для вибору кольору літер та їх фону:

```
textcolor (YELLOW);        /* встановлення жовтого кольору літер */
textbackground (BLUE);     /* встановлення темно-синього фону */
```

Можна оголошувати змінні, що будуть мати визначений переліковий тип. Наприклад, у разі оголошення:

```
enum work_days  day;
```

змінна day зможе набувати значення кожної із констант переліку work_days:

```
day=THU;                   /* день - четвер */
if (day>WED)               /* якщо це друга половина тижня, то */
  . . .                    /* виконати певні дії */
```

Змінні перелікового типу можна оголошувати окремо (як у попередньому прикладі для day) або разом з оголошеннями переліків:

```
enum result {NO, YES} res1, res2;
```

Перелікові змінні призначені для збереження цілочислових значень, присвоєних їм через іменовані константи. Якщо потрібно відобразити значення такої змінної, то воно повинно виводитись як дане цілого типу. Якщо ж бажано вивести символьне (текстове) значення змінної, то можна скористатись, наприклад, таким прийомом:

```
if (res1==YES)
  puts ("Так");            /* виведення варіанта YES */
else
  puts ("Ні");             /* виведення варіанта NO */
```

Для виведення текстових еквівалентів значень елементів переліку найчастіше використовують оператор switch або спеціальні масиви символьних рядків, кожен з яких містить текст, що розкриває відповідну іменовану константу. Приклад використання оператора switch для виведення словесних найменувань елементів переліку наведено в розділі 6, а нижче подано приклад виведення значення перелікової змінної через застосування додаткового масиву найменувань днів тижня.

```
/*  Виведення значення змінної перелікового типу  */
enum WDAYS  { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
char* day_name[]={ "неділя", "понеділок", "вівторок", "середа",
                   "четвер", "п\'ятниця", "субота" };
enum WDAYS  wday=Fri;

    printf (" Цей день - %s.\n", day_name[wday]);
```

У результаті виконання наведеного фрагмента програми буде виведено повідомлення:

```
Цей день - п'ятниця.
```

Слід пам'ятати, що компілятор не контролює діапазон значень перелікової змінної. Присвоєння:

```
wday=8;        /* 8 - не входить у набір значень enum WDAYS */
```

буде синтаксично коректним і змінна `wday` набуде значення `8`. Оскільки це значення перевищує граничне для типу `enum WDAYS`, то наступні операції зі змінною `wday` можуть призвести до хибної роботи програми.

Запитання та завдання для самоконтролю

1. Які типи даних називають базовими, а які похідними?
2. Якими є діапазони значень змінних, що мають тип `int`, і змінних з типом `unsigned`?
3. Який обсяг оперативної пам'яті займають дані, що мають тип `unsigned long`? Яким є діапазони їх значень?
4. Який тип отримає кожна із перелічених констант: `24`, `04382`, `0xf055`, `-550`, `70000`, `108400u`, `3281`, `'5'`, `'\n'`?
5. Як зберігаються від'ємні цілі числа? Яким буде двійковий код числа -15?
6. Який заголовний файл треба підключити до програми, щоб використовувати константи граничних значень цілих типів? А для дійсних типів?
7. Що називають мантисою і порядком дійсного числа? Як зберігаються в оперативній пам'яті дійсні числа?
8. Чим відрізняються між собою дійсні типи `float`, `double` та `long double`?
9. Наведіть приклади декількох дійсних констант, значення яких дорівнюють числу 68,025.
10. Оголосіть змінні, призначені для збереження значень:
    1) кількості учасників шахового турніру;
    2) кількості жителів європейської держави;
    3) середнього віку жителів європейської держави;
    4) кількості друкованих знаків у цьому посібнику;
    5) сумарної вартості проданих кондитерських виробів;
    6) віддалі до заданої планети сонячної системи.
11. Як ініціалізують змінні в оголошеннях? Які значення можна використовувати для ініціалізації змінних?
12. Оголосіть константну змінну, значення якої дорівнює поточному року.
13. Оголосіть перелік, елементи якого відповідають місяцям року.
14. Як оголошуються змінні перелікового типу? Які значення можна присвоювати цим змінним? Наведіть приклади, використовуючи перелік з п. 13. Чи здійснюється контроль за правильністю значень, які присвоюються змінним перелікового типу?

Розділ 4

ВИРАЗИ ТА ОПЕРАЦІЇ

У цьому розділі:

- Поняття виразу, операнда та операції
- Повна таблиця операцій мови С, класифікація операцій
- Порядок обчислення значення виразу
- Характеристика арифметичних операцій
- Операції над бітами даних: інвертування розрядів, побітові операції AND, XOR та OR, операції порозрядного зсування
- Операції порівняння та логічні операції, їх властивості у виразах
- Особливості операцій присвоєння: звичайне присвоєння, комбіновані присвоєння, префіксний та постфіксний інкремент/декремент
- Операції умовного вибору операнда та визначення розміру операнда
- Правила узгодження типів у виразах – арифметичні перетворення типів і перетворення типів в операціях присвоєння
- Стандартні математичні функції із заголовного файла math.h
- Макропідстановки з параметрами

Мова С вирізняється поміж інших мов програмування наявністю потужних і гнучких засобів формування та запису виразів. *Виразом* називають послідовність операндів, об'єднаних знаками операцій та круглими дужками, яка має певне значення одного з допустимих типів. В цьому означенні *операнди* – це об'єкти, над якими виконуються операції, а *операції* задають дії, що виконуються над операндами.

Наприклад, виразами є наступні записи:

```
d+b*c-f/5
0.5*(z+w)>=(z-u)/(w-u)
kor=sin(x)+2*exp(x-0.7)
n==1 ? "рядок" : "рядки"
```

Операндами виразів можуть бути:

- константи;
- змінні;
- первинні вирази – елементарні підвирази даного виразу.

Операнди більшості операцій мови С повинні бути скалярними даними, тобто вони можуть мати арифметичний, переліковий або вказівниковий тип.

Мова С використовує великий і потужний набір вбудованих операцій (їх аж 46!), що має визначальний вплив на лаконізм і компактність запису С-програм. Повний перелік операцій мови С подано в табл. 4.1.

Операції можна класифікувати за різними ознаками. Зокрема, за кількістю операндів, що вступають в операцію, виділяють:

- *унарні* операції – застосовуються до одного операнда;
- *бінарні* – в операцію вступають два операнди;
- *тернарні* – в операції використовуються три операнди (мова С має одну тернарну операцію, яка називається умовною).

За видом дій, що виконуються над операндами, операції мови С поділяють на:

- арифметичні;
- порівняння (відношення);
- логічні;
- порозрядні (побітові);
- присвоєння;
- інші.

За *старшинством* (*пріоритетом*) операції поділяють на 16 рівнів (див. табл. 4.1). Найвищий пріоритет мають т. зв. *первинні* операції, а найнижчий (перший рівень старшинства) – операція послідовних обчислень "кома".

Таблиця 4.1

Операції мови С

Рівень старш.	Знак	Зміст операції	Тип	Асоціативність
16	( )	Звертання до функції	Первинні	Зліва направо
	[ ]	Виділення елемента масиву через індекс		
	.	Виділення елемента структури		
	->	Виділення елемента структури через вказівник		
15	++	Постфіксний інкремент (збільшення на 1)	Присвоєння	—
	--	Постфіксний декремент (зменшення на 1)		
14	-	Зміна знака	Різнотипні унарні	Справа наліво
	++	Префіксний інкремент (збільшення на 1)		
	--	Префіксний декремент (зменшення на 1)		
	!	Логічне заперечення		
	~	Побітове заперечення		
	&	Визначення адреси		
	*	Звертання через вказівник (адресу)		
	(*тип*)	Перетворення до заданного типу		
	sizeof	Визначення розміру операнда в байтах		

13	*	Множення	Арифметичні	Зліва направо
	/	Ділення		
	%	Остача від ділення		
12	+	Додавання		
	-	Віднімання		
11	<<	Зсування вліво	Зсування (побітові)	
	>>	Зсування вправо		
10	<	Менше	Порівняння (відношення)	
	<=	Не більше		
	>	Більше		
	>=	Не менше		
9	==	Дорівнює		
	!=	Не дорівнює		
8	&	Побітове AND	Порозрядні (побітові)	
7	^	Побітове XOR		
6	|	Побітове OR		
5	&&	Логічне І	Логічні	
4	||	Логічне АБО		
3	? :	Вибір операнда за умовою	Умовна	Справа наліво
2	=	Звичайне присвоєння	Присвоєння	
	*=	Множення з присвоєнням		
	/=	Ділення з присвоєнням		
	%=	Остача від ділення з присвоєнням		
	+=	Додавання з присвоєнням		
	-=	Віднімання з присвоєнням		
	>>=	Зсування вправо з присвоєнням		
	<<=	Зсування вліво з присвоєнням		
	&=	Побітове AND з присвоєнням		
	|=	Побітове XOR з присвоєнням		
	^=	Побітове OR з присвоєнням		
1	,	Послідовні обчислення	Кома	Зліва направо

В англомовній літературі з програмування для терміну *операція* (в сенсі *знак операції*) здебільшого використовують слово *operator*, яке може бути перекладене і як знак операції, і як оператор. Ми будемо дотримуватись більш усталених для нашої літератури термінів *операція* та *знак операції*, але в багатьох перекладних виданнях можна зустріти в тому ж контексті термін *оператор* (зокрема, в перших перекладах відомої книги Б. Кернігана та Д. Ритчі "The C Programming Language" [10]).

4.1. Порядок виконання операцій

Порядок обчислення значення виразу базується на таких основних правилах:

- з двох сусідніх операцій першою виконується та, рівень старшинства якої вищий;
- операції однакового рівня старшинства виконуються зліва направо або справа наліво залежно від асоціативності, встановленої для даної групи операцій (див. табл. 4.1);
- порядок виконання операцій можна змінити за допомогою круглих дужок `()` – вираз у дужках обчислюється першим і розглядається як операнд зовнішньої операції.

Наведемо три приклади виразів, в яких цифрами під знаками операцій позначено порядок їх виконання:

```
res = k + m - a / 3
      4   1   3   2

x <= 5.7 * (r - y) / (r - 0.2)
   5       2     1     4      3

n > 0 && (c == '1' || c == '0')
   1    5       2        4      3
```

Водночас потрібно пам'ятати, що мова С не встановлює порядку звертання до операндів у процесі виконання операції. Тобто, у виразі

```
w = (a + b) * (a - b)
```

операцію множення `*` гарантовано буде виконано перед операцією присвоєння `=`, а ще перед цим будуть обчислені вирази в дужках. Але який з двох операндів-виразів: `(a+b)` чи `(a-b)` буде обчислено першим – не визначено (це залежить від апаратно-програмних особливостей реалізації арифметичних операцій у даному середовищі). Тому порядок обчислення наведеного виразу може бути таким:

```
w = (a + b) * (a - b)
    4     1     3     2
```

або таким:

```
w = (a + b) * (a - b)
    4     2     3     1
```

Звісно, це ніяк не вплине на кінцеве значення заданого виразу. Проте можливі випадки, коли значення виразу залежить від порядку звертання до операндів. Наведемо приклад такого оператора:

```
d = c+twice(&c);
```

Нехай перед виконанням даного оператора `c` дорівнює 5, а функція `twice()` збільшує вдвічі значення змінної `c`, адреса якої передається їй як параметр, і повертає нове значення. Тоді значення виразу, що присвоюється `d`, може дорівнювати `5+10=15`, якщо в даній системі першим вибирається лівий операнд, а потім – правий; або `10+10=20`, якщо спочатку обчислюється правий операнд, тобто відбувається виклик функції `twice(&c)`, яка змінює значення `c`. Наведений приклад доводить, що треба уникати виразів, результати обчислення яких можуть бути неоднозначними. Приміром, у даному прикладі можна було скористатись двома операторами присвоєння:

```
d=c;     d=d+twice(&c);        /* для першого випадку */
```

або

```
d=twice(&c);     d=d+c;        /* для другого випадку */
```

У записах складних виразів можна, а часто – доцільно, використовувати роздільчі дужки `()` для підвищення читабельності виразу, навіть якщо згідно зі синтаксисом мови вони й не є необхідними. Поданий нижче приклад умовного виразу значно легше читати, коли його підвирази виділені дужками:

```
def = (c>d) ? (c-d) : (d-c)
```

Далі розглянемо детальніше операції основних класифікаційних груп.

4.2. Арифметичні операції

До арифметичних належать наступні операції (список складено згідно з порядком старшинства операцій):

- `-` (зміна знака операнда) – унарна операція;

  бінарні операції:
- `*` (множення), `/` (ділення), `%` (обчислення остачі від ділення);
- `+` (додавання), `-` (віднімання).

Операції `*`, `/`, `-` та `+` виконуються над даними всіх числових типів (цілих і дійсних), а операція `%` (обчислення остачі від ділення) потребує двох цілочислових операндів.

Результат виконання кожної з арифметичних операцій над двома операндами цілого типу завжди є цілим.

Нехай в програмі оголошено:

```
int  b=7, g=3;
```

тоді наступні вирази матимуть такі значення:

```
-b ⇒ -7     b+g ⇒ 10     g-b ⇒ -4     b*g ⇒ 21     b*-g ⇒ -21
b/g ⇒ 2     g/b ⇒ 0      b%g ⇒ 1      g%b ⇒ 3
```

Часто потрібно визначити дійсну частку від ділення двох цілих операндів: `1` поділити на `7` або цілочислову змінну `c` поділити на `3` тощо. Якщо використати вирази `1/7` та `c/3`, то результат їх буде цілим, а не дійсним числом: значення першого виразу дорівнює `0`, а другого – цілій частині третини `c`. Щоб частка була дійсною, дійсним має бути значення хоча б одного з двох операндів (детальніше питання взаємоузгодження типів операндів розглядаються далі). Тому зазначені вирази слід записати, наприклад, так: `1.0/7.0` або `(double)c/3`. В останньому виразі використано операцію явного перетворення типу (*тип*). Оскільки її старшинство вище, ніж старшинство операції ділення, то спочатку значення `c` перетворюється до форми дійсного числа з типом `double`, а наступне ділення на `3` обчислює частку, яка теж матиме тип `double`.

4.3. Порозрядні операції

Порозрядні операції (їх ще називають побітовими чи просто бітовими) виконуються тільки над операндами цілих типів. До них належать (згідно зі старшинством):

- ~ (побітова інверсія) – унарна операція;

  бінарні операції:
- << (зсування вліво), >> (зсування вправо);
- & (побітове AND чи побітова кон'юнкція);
- ^ (побітове XOR чи додавання за модулем два);
- | (побітове OR чи побітова диз'юнкція).

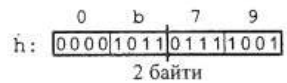
Використання порозрядних операцій дає змогу оперувати з окремими бітами даних, перевіряти або змінювати їх значення, що наближає програмування мовою С до можливостей асемблера. Розглянемо названі операції.

Побітова інверсія. Унарна операція ~ виконує інвертування двійкових розрядів числа: всі біти, що мали значення 0, стають 1 і, навпаки, всі 1 стають 0.

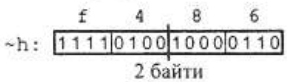
Нехай у програмі оголошено:

```
unsigned int  h=0xb79;
```

Яким буде значення виразу ~h? Щоб пояснити відповідь, розглянемо внутрішній код змінної h.



Двійкове значення виразу ~h є таким:



що відповідає шістнадцятковому числу 0xf486 та десятковому значенню 62589.

Звернемо увагу, що інвертуються всі розряди числа, включаючи знаковий. Якби змінна h мала тип int, а не unsigned, то результат інвертування розглядався б як від'ємне число, оскільки знаковий біт став 1.

Таблиці істинності булевих операцій. Три бінарні порозрядні операції, що позначаються знаками: &, ^ та |, реалізують операції, які належать до класу *булевих* (чи операцій *алгебри логіки*). Їх називають відповідно: AND, XOR та OR. Операції виконуються окремо над кожною парою бітів, що записані в однакових за номером розрядах двох операндів, які вступають у порозрядну операцію. Значення результатів цих операцій (т. зв. *таблиці істинності*) згруповані в табл. 4.2.

Побітове AND. З табл. 4.2 видно, що у випадку виконання над цілочисловими операндами a і b порозрядної операції &, у двійковому коді результату операції одиниці будуть тільки в тих розрядах, у яких обидва біти a_i та b_i (де i – номер розряду) є одиницями. Для прикладу розглянемо, чому буде дорівнювати результат операції a&b,

Таблиця 4.2

Значення порозрядних операцій

Розряди		Значення результату операції		
a_i	b_i	AND	XOR	OR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	1

якщо `a=0x183e` і `b=0x3f02`. Запишемо двійкові зображення операндів і результату:

```
0001|1000|0011|1110   – перший операнд (a)
&
0011|1111|0000|0010   – другий операнд (b)
---------------------
0001|1000|0000|0010   – результат
```

Отже, значення виразу `a&b` дорівнюватиме шістнадцятковому числу `0x1802 (6146)`.

Операцію `&` називають накладанням маски, оскільки вона дає змогу виділити потрібні розряди цілочислового операнда. Нехай треба перевірити, чи хоч один із трьох молодших розрядів числа, записаного в змінну `reg`, що має тип `unsigned int`, дорівнює одиниці. Таку перевірку можна ефективно виконати, наклавши на `reg` маску, в якій всі біти, крім трьох останніх, будуть нульовими: `0x0007`. У двійковому значенні результату всі старші біти дорівнюватимуть нулю, а три молодші біти збережуть ті значення, які вони мали в `reg`. Перевірка може виглядати так:

```
/*  Перевірка трьох молодших розрядів цілого числа  */
   if (reg & 0x0007 > 0)
       . . .                    /* дії для випадку, коли є 1 */
   else
       . . .                    /* дії, коли всі три розряди 0 */
```

Побітове OR. Операція `|` встановлює одиницю в тих бітах значення результату, в яких хоча б один із операндів мав одиницю. Наприклад, результат операції `a|b` матиме значення, що дорівнює шістнадцятковому числу `0x3f3e (16190)`:

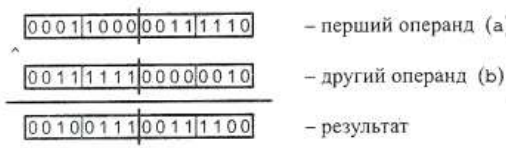
```
0001|1000|0011|1110   – перший операнд (a)
|
0011|1111|0000|0010   – другий операнд (b)
---------------------
0011|1111|0011|1110   – результат
```

Операцію `|` найчастіше використовують для встановлення в 1 потрібних розрядів числових даних. Приміром, щоб встановити в одиницю четвертий і п'ятий розряди змінної `reg`, треба виконати присвоєння:

```
reg = reg|0x30;
```

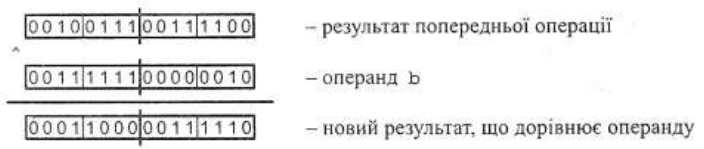
тобто накласти на змінну `reg` OR-маску, в якій у четвертому і п'ятому розрядах будуть одиниці (розряди числа нумерують справа наліво, починаючи з нуля).

Побітове XOR. Порозрядна операція `^` виконує над кожною парою бітів булеву операцію XOR (її ще називають додаванням за модулем два). У значенні результату одиниці будуть у тих розрядах, які в операндів не збігаються (див. табл. 4.2). Тому результат операції `a^b` буде таким:



що відповідає числу `0x273c` (10044).

Операція `^` має цікаву властивість: якщо її виконати повторно над результатом і одним із початкових операндів, то отримаємо значення другого операнда:

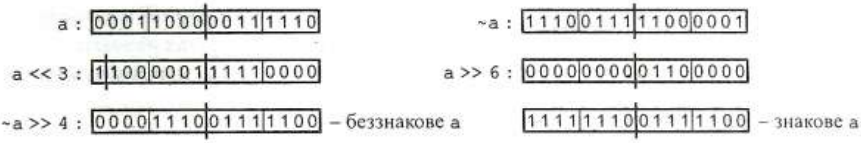


Операції зсування. Операції `<<` та `>>` зсувають усі розряди цілочислового операнда, заданого зліва від знака операції, на кількість бітів, вказану значенням, записаним справа від знака операції. Перша з цих операцій виконує зсування вліво, а друга – вправо.

У разі зсування вліво, старші біти, які виходять за межі розрядної сітки числа, втрачаються, а на місце молодших бітів, що звільнились після зсування, заносяться нулі. Лівостороннє зсування числа на `n` розрядів часто використовують для швидкого множення даного на 2^n. У процесі зсування беруть участь усі розряди числа, в т. ч. і біт знака. Тому зсування вліво знакових чисел може призвести до зміни знака операнда через потрапляння в біт знака інверсного значення.

У разі зсування операнда вправо втрачаються молодші розряди числа, які зсуваються за межі розрядної сітки. Заповнення вивільнених старших розрядів залежить від типу даного: якщо операнд беззнаковий (оголошений з модифікатором `unsigned`), то вільні старші розряди заповнюються нулями, якщо ж праворуч зсувається знакове число, то для заповнення вільних розрядів використовується біт знака. Таким чином, при зсуванні вправо зберігається знак числа, а результат зсування на `n` розрядів рівнозначний до цілочислового ділення даного на 2^n.

Проілюструємо результати операцій зсування прикладами двійкових кодів значень наступних виразів: `a<<3`, `a>>6` та `~a>>4`.



Старший біт результату зсування `a<<3` дорівнює двійковій одиниці, тому інтерпретація значення цього виразу залежить від того, чи змінну `a` було оголошено як знакову, чи як беззнакову. Якщо `a` беззнакова змінна (тип `unsigned int`), то результат зсування теж беззнакове число, шістнадцяткове значення якого дорівнює `0xc1f0`, що відповідає десятковому числу `49648`. Якщо ж `a` оголошено як знакову цілу (тип `int`), то результат зсування буде розглядатись як знакове дане, в якому найстарший біт визначає знак. Так як в цей біт потрапила 1, то число вважається від'ємним і записаним у доповнювальному коді. Тому той самий двійковий код тепер відповідає числу `-15887`. При зсуванні значення `a` вправо вільні старші розряди заповнюються 0, незалежно від типу змінної `a`, оскільки біт знака `a` заповнений 0. А ось результат зсування вправо значення `~a` залежить від типу операнда. Якщо тип `a` (тим самим і тип `~a`) беззнаковий, то вільні старші розряди заповнюються нулями, якщо ж `a` знакова змінна, то вивільнені розряди заповнюються бітом знака, який для `~a` дорівнює 1.

4.4. Операції порівняння

Всі шість операцій порівняння (їх ще називають операціями відношення) бінарні:

- `>` (більше), `>=` (не менше), `<` (менше), `<=` (не більше);
- `==` (дорівнює), `!=` (не дорівнює).

Насамперед звернемо увагу на знаки операцій "дорівнює" `==` і "не дорівнює" `!=`. Поширеною помилкою є використання знака присвоєння `=` замість знака рівності. Оскільки присвоєння це також операція, що повертає присвоєне значення, то компілятор не фіксує помилки в таких виразах.

Компілятор Borland C видає попередження, якщо у виразі, записаному як умова (зокрема в операторах `if`, `for`, `while`), останньою виконується операція присвоєння, але стандартно не переривається процесу компіляції програми. Тому доцільно завжди перевіряти попередження компілятора, щоб не пропустити можливі помилки.

Іншою особливістю операцій порівняння є значення результату. В мові С істиною (TRUE) вважається довільне числове значення, що не дорівнює нулю, а хибність (FALSE) позначається `0`. Тому всі операції порівняння повертають числове значення: ненульове значення (здебільшого `1`), якщо результат порівняння істинний, та `0`, якщо результат операції порівняння хибний.

Нехай в програмі оголошено:

```
double  teta=1.634;
```

Тоді значення наступних виразів будуть такими:

`teta>0.5` ⇒ ≠0 (істинне); `2*teta<=3` ⇒ `0` (хибне).

Третьою особливістю операцій порівняння є поділ їх на два рівні старшинства. В записі наступного виразу використано ту властивість, що операція рівності == має нижчий пріоритет, ніж дві інші: < та <= (цифрами позначено порядок виконання операцій).

```
x < xmin == y <= ymax
1       3     2
```

Результат усього виразу буде істинним (ненульовим), коли обидві операції порівняння (< та <=) будуть одночасно істинними або хибними. Запис такої умови з використанням логічних операцій буде вдвічі довшим.

Звернемо також увагу на можливі помилки у виразах з послідовними операціями порівняння. Нехай треба записати вираз, який буде істинним, якщо точка $A(x)$, розміщена на числовій осі, потрапляє в інтервал $[c, d]$, тобто, коли $c \le x \le d$. Якщо в програмі цю умову записати подібним виразом порівняння:

```
c <= x <= d                    /* !! небезпечна помилка !! */
```

то синтаксично цей вираз буде правильним (компілятор не зафіксує помилки), але семантично (за змістом) записаний вираз зовсім не відповідає умові задачі. Згідно з асоціативністю операцій порівняння він інтерпретується так: `(c <= x) <= d`. Спочатку виконується порівняння `c <= x`, результатом операції буде `0` або `1`. Потім це значення порівнюється з `d`. Якщо значення `d` не менше за `1`, то записаний вираз буде істинним, незалежно від значень `x` та `c`. Тому, щоб вираз був правильним, треба застосувати операцію логічного "і":

```
x >= c && x <= d               /* правильний вираз */
```

4.5. Логічні операції

У мові С застосовують три логічні операції:

- `!` (логічне НЕ – заперечення) – унарна операція;

бінарні операції:

- `&&` (логічне і);
- `||` (логічне АБО).

Операндами логічних операцій є логічні дані, а саме: істина (TRUE) – в мові С це довільне числове значення, яке не дорівнює `0`, та хибність (FALSE), що позначається `0`. Найчастіше операндами логічних операцій бувають вирази з операціями порівняння.

Операція логічного заперечення `!` змінює логічне значення на протилежне. Вираз:

```
!(x>a && x<b)
```

буде істинним, коли $x \notin (a, b)$. Цей вираз можна було б записати і так:

```
x<=a || x>=b
```

Операцію заперечення часто використовують у такому контексті: замість операції порівняння `x == 0` записують `!x`.

Результат бінарної операції `&&` буде істинним (≠0) тільки в тому разі, коли обидва операнди мають ненульове значення (обидва операнди є істинними). Щоб істинним був результат бінарної операції `||`, достатньо істинності хоча б одного з операндів.

Приклад логічного виразу:

```
sym != '\n' && sym != '\t' || k == KMAX
```

Цей вираз істинний за умови, що значення змінної `sym` не є ні символом нового рядка `'\n'`, ні символом табуляції `'\t'` або, якщо значення `k` дорівнює `KMAX`. Старшинство операцій мови С дає змогу записати такий вираз без використання дужок, оскільки операції порівняння мають вищий пріоритет, ніж логічні.

Логічні вирази в С не обов'язково повинні обчислюватись повністю. Як тільки значення виразу стає однозначним, подальші обчислення припиняються. Приміром, якщо значення змінної `sym` із попереднього прикладу не є ні символом `'\n'`, ні символом `'\t'`, то весь вираз буде істинним, незалежно від результату порівняння `k == KMAX`, тому остання операція може не виконуватись взагалі.

Формуючи логічні вирази, слід обов'язково враховувати властивість неповних обчислень. Наприклад, у виразі:

```
k < KMAX && ++k >= ksum
```

правий операнд операції `&&` не буде обчислюватись, якщо значення виразу `k < KMAX` буде хибним (дорівнюватиме `0`), отже, операція інкремента `++k` може не відбутись. Тому в записах логічних виразів першими доцільно вказувати умови, які змінюються найчастіше, тобто ті, які необхідно перевіряти щоразу.

Ще одна специфічна властивість логічних операцій: їх лівий операнд завжди обчислюється першим, а правий – другим (нагадаємо, що для арифметичних та інших операцій такий порядок не є обов'язковим). Ця властивість дає змогу записувати популярні в С-програмах вирази, в яких відразу виконується введення даних і їх перевірка. Наприклад, наступний вираз перевіряє, чи введений символ є шістнадцятковою цифрою:

```
(ch = getchar()) >= '0' && ch <= '9' ||
 ch >= 'A' && ch <= 'F' || ch >= 'a' && ch <= 'f'
```

Дужки, якими в цьому виразі охоплено операцію присвоєння, обов'язкові, бо пріоритет присвоєння нижчий, ніж наступної операції порівняння. Інші дужки у виразі не потрібні, хоча їх можна було б використати для наочнішого виділення складових частин довгого виразу.

4.6. Операції присвоєння

Група операцій присвоєння найбільш численна – їх усього 15:

унарні операції:

- `++` (постфіксний/префіксний інкремент), `--` (постфіксний/префіксний декремент);

бінарні операції:

- = (звичайне присвоєння), *= (множення з присвоєнням), /= (ділення з присвоєнням), %= (остача від ділення з присвоєнням), += (додавання з присвоєнням), -= (віднімання з присвоєнням), &= (побітове AND з присвоєнням), ^= (побітове XOR з присвоєнням), |= (побітове OR з присвоєнням), <<= (зсування вліво з присвоєнням), >>= (зсування вправо з присвоєнням).

Операції присвоєння відрізняються від інших операцій тим, що вони змінюють значення свого операнда: у разі виконання бінарних операцій лівий операнд набуває значення виразу, записаного справа від знака присвоєння =, а в разі унарних операцій змінюється початкове значення єдиного операнда даної операції. Тому операнд, якому присвоюється значення, повинен бути т. зв. *l-операндом*: змінною або розадресованим вказівником, тобто він має задавати адресу ділянки оперативної пам'яті, куди буде записане значення, яке присвоюється.

Термін *l-операнд* (або *l-значення*, в оригіналі – *l-value* – від слів *left side value*) введено авторами мови С для позначення об'єктів, які можуть записуватись у лівій частині операцій присвоєння. Найчастіше *l*-операндом є ім'я змінної, яке неявно задає її адресу в пам'яті комп'ютера. Іншим видом *l*-операндів є вирази, що здійснюють звертання до оперативної пам'яті через адресу, задану вказівником або константним значенням. Застосовувати такі *l*-значення будемо після того, як розглянемо вказівникові типи та властивості адресної арифметики.

Результатом операції присвоєння є значення, яке набуває змінна (*l*-операнд) після присвоєння. Це значення може використовуватись як операнд у наступних операціях, для яких присвоєння є підвиразом.

Ще однією особливістю бінарних операцій присвоєння є їх асоціативність – присвоєння виконуються справа наліво.

Звичайне присвоєння. Найпоширенішою з операцій є звичайне присвоєння =, в якому лівий операнд набуває значення виразу, записаного справа від знака присвоєння:

```
polin = a*x*x + b*x + c
```

У наведеному виразі змінна `polin` набуває значення виразу `a*x*x + b*x + c`. Присвоєне значення є одночасно і значенням усього записаного вище виразу.

Оскільки в мові С присвоєння є операцією, то можна записати вираз, який буде складатись із декількох присвоєнь:

```
u = w = z = beg
```

Асоціативність операцій присвоєння (справа наліво) визначає порядок виконання присвоєнь. У результаті всі три змінні `z`, `w` і `u` почергово набудуть значення змінної `beg`, це значення буде також загальним значенням усього виразу.

Бінарні присвоєння за старшинством стоять у кінці списку, нижчий пріоритет має тільки операція "кома". У переважній більшості виразів з присвоєннями власне присвоєння повинно виконуватись останнім, тому низький пріоритет даної операції дає змогу уникнути використання дужок `()`. Проте, якщо присвоєння є підвиразом більш загального виразу, то його слід охоплювати дужками. Наприклад, у виразі

```
(dif=newk-oldk) > RANGE
```

змінна `dif` спочатку набуває значення різниці `newk-oldk`, а потім це значення порівнюється з граничним, заданим через `RANGE`.

Важливим для формування виразів у цілому, і операцій присвоєння зокрема, є питання узгодження і перетворення типів операндів. Ці питання детально розглядаються далі. Зараз лише зазначимо, що в операціях присвоєння тип виразу справа від знака `=` перетворюється до типу змінної (*l*-операнда), якій присвоюється значення. Якщо типи змінної та виразу несумісні, то компілятор фіксує помилку.

Комбіновані присвоєння. Зручними в записі та ефективними в реалізації є комбіновані присвоєння: `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`. Вони не тільки спрощують запис виразу та роблять його більш наочним, але й скорочують процес обчислення значення виразу.

За своїми діями комбіновані присвоєння є окремим випадком звичайних присвоєнь. Вираз

```
x ⊕= expr
```

є скороченим записом операції присвоєння:

```
x = x ⊕ expr
```

тут `⊕` – один зі знаків, що застосовуються в комбінованих присвоєннях; `x` – змінна (*l*-операнд); `expr` – певний вираз (підвираз), значення якого вступає в операцію `⊕`.

Подані нижче приклади пар операторів виконують присвоєння, значення результатів яких збігаються:

```
sum += number;                  sum = sum + number;
prod *= faw + 6;                prod = prod * (faw + 6);
flag <<= count & 0x7;           flag = flag << (count & 0x7);
```

Останнім наведемо цікавий приклад, у якому використано ланцюжок комбінованих присвоєнь. Нехай у програмі оголошено змінні й записано оператор:

```
  int  c = 5, d = 20;
c ^= d ^= c ^= d;
```

Допитливим читачам пропонуємо самостійно дати відповідь, яким буде результат виконання цього оператора. Тим же, хто ще відчуває труднощі в роботі з порозрядними операціями та комбінованими присвоєннями, допоможемо поясненнями. Першим у виразі буде виконане останнє присвоєння `c^=d` – у змінну `c` буде записано результат операції побітового XOR значень `c` і `d` (змінна `c` набуде значення `17`). Наступне присвоєння змінює значення `d` – воно дорівнюватиме результату операції `^` над початковим значенням `d` і результатом попередньої операції. Раніше вже відзначалося, що повторне виконання порозрядної операції `^` відновлює значення іншого операнда, тому в `d` буде записано початкове значення `c` (`d` дорівнюватиме `5`). Третє виконання операції `^=` над змінною `c`, в якій зберігається результат першого побітового XOR, та новим значенням змінної `d`, що дорівнює початковому значенню `c`, відновлює

початкове значення d і заносить його в змінну c (c набуває значення 20). Отже, в результаті виконання всього оператора змінні c і d міняються значеннями. Важливо, що цей обмін відбувається без застосування додаткової змінної.

Унарні присвоєння (інкремент і декремент). У С-програмах широко застосовують унарні операції присвоєння ++ та --. Перша з них, яка називається *інкрементом*, збільшує значення свого операнда на 1. Друга операція називається *декрементом*, вона зменшує на 1 значення заданого операнда. Мова С використовує дві форми операцій інкремента та декремента: префіксну, коли знак ++/-- записується перед операндом, і постфіксну, коли знак операції вказується після операнда. Нагадаємо, що операнд унарної операції присвоєння повинен бути змінною або іншим *l*-значенням.

Якщо унарне присвоєння використовується як окремий оператор, то обидві форми: префіксна та постфіксна рівнозначні. Всі чотири наступних оператори виконують ту ж саму дію: збільшують на 1 значення змінної k:

```
++k;        k++;        k += 1;        k = k + 1;
```

Відповідно зменшити на 1 значення k можна кожним з наступних операторів:

```
--k;        k--;        k -= 1;        k = k - 1;
```

Різниця між префіксною і постфіксною формами операцій інкремента та декремента проявляється, коли вони використовуються як операнди інших операцій. Префіксний інкремент/декремент встановлює, що спочатку значення змінної збільшується чи, відповідно, зменшується на 1, а вже потім вона вступає в іншу операцію. Якщо ж використовується постфіксна форма, то змінна спочатку віддає у вираз своє значення, а вже потім збільшується/зменшується на 1.

Проілюструємо сказане двома прикладами програм, в першому з яких у вираз, що визначає умову виконання циклу, входить операція префіксного інкремента, а в другому прикладі – операція постфіксного інкремента.

```
/*****************************************/
/*   Застосування префіксного інкремента  */
/*****************************************/

#include <stdio.h>
int  main(void)
{
    int  k=1,  max=5;
  do {
    printf(" k=%d  ", k);
  } while (++k < max);                    /* умова виконання циклу */
  printf ("\t (значення k після циклу - %d) \n",  k);
  return 0;
}
```

У процесі виконання цієї програми на екран буде виведено:

```
k=1   k=2   k=3   k=4     (значення k після циклу - 5)
```

Циклічні дії наведеної програми виконуються `max-1` разів. Якщо ж в умові циклу використати постфіксний інкремент, то результат виконання буде іншим.

```
/*****************************************/
/*   Застосування постфіксного інкремента  */
/*****************************************/
#include <stdio.h>
int main (void)
{
     int  k=1,  max=5;
   do {
     printf(" k=%d  ", k);
   } while (k++ < max);                    /* інша умова виконання циклу */
   printf("\t (значення k після циклу - %d) \n",  k);
   return 0;
}
```

Результат виконання:

```
k=1   k=2   k=3   k=4   k=5    (значення k після циклу - 6)
```

Цикл повторюється `max` разів, оскільки у виразі умови спочатку виконується порівняння `k < max`, а вже потім операція інкремента `k++`. Перед завершенням циклу значення `k` збільшується останній раз і набуває значення `6`.

4.7. Умовна операція

Цю операцію називають також операцією вибору операнда. Вона виконується над трьома операндами і записується так:

операнд_1 ? *операнд_2* : *операнд_3*

Якщо значення виразу *операнд_1* не дорівнює `0` (істинне), то результатом умовної операції є значення виразу *операнд_2*, якщо ж *операнд_1* дорівнює `0` (хибний), то результатом операції є значення виразу *операнд_3*.

Наступний вираз визначає більшу з двох заданих величин:

```
gr >= 100 ? gr : 100
```

а другий приклад є виразом, що повертає модуль (абсолютну величину) змінної `x`:

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

У багатьох випадках умовною операцією можна замінити умовний оператор. Наприклад, обчислення y за заданою умовою:

$$y = \begin{cases} 2 \cdot \sqrt{z}, & \text{якщо } z \geq 5 \\ z^2 - z, & \text{в інших випадках} \end{cases}$$

можна запрограмувати так:

```
y = z >= 5 ? 2*sqrt(z) : z*z-z;
```

Раціональне старшинство операцій мови С дає змогу не використовувати в записі попереднього оператора круглих дужок. Нагадаємо: присвоєння має нижчий пріоритет, ніж умовна операція, яка, в свою чергу, поступається за старшинством усім іншим.

4.8. Операція розміру `sizeof`

Унарна операція `sizeof` повертає обсяг пам'яті, яку займає або потребує операнд цієї операції, тобто *розмір* операнда. Розмір визначається в байтах.

Операція `sizeof` має дві стандартні форми:

`sizeof (` *тип* `)`

`sizeof` *ім'я_змінної*

Якщо операнд є найменуванням типу, то його треба записувати в круглих дужках, якщо ж операнд є іменем змінної, то дужки не потрібні, хоча їх використання не вважається помилкою.

Наведемо приклади. Нехай у програмі оголошено:

```
double x=4.25;
```

тоді обидва вирази:

```
sizeof x      та      sizeof (double)
```

повернуть значення `8`. Наступний вираз:

```
sizeof(unsigned)*N
```

визначає обсяг пам'яті, необхідної для збереження масиву з `N` елементів, що мають тип `unsigned int`. Звернемо увагу, що значенням виразу

```
sizeof(N*x)
```

є розмір типу, який має підвираз `N*x`, тобто розмір типу `double` – `8`.

4.9. Узгодження типів у виразах

Операнди бінарних операцій мови С можуть бути числовими (арифметичними) даними або вказівниками. Операції над вказівниками розглядатимемо в розділі 7. Зараз мова йтиме тільки про арифметичні операнди.

Якщо обидва операнди бінарної операції мають однаковий тип, то такий самий тип матиме результат виконання даної операції (виняток становлять операції порівняння, результат яких завжди цілий). Проте операнди, над якими виконується бінарна операція, можуть мати й різні арифметичні типи. У цьому випадку перед виконанням операції компілятор перевіряє сумісність операндів і встановлює для них спільний тип. Цей тип буде також типом результату виконання операції. Існують дві схеми узгодження типів операндів у виразах:

1) арифметичні перетворення типів;

2) перетворення типів в операціях присвоєння.

Арифметичні перетворення типів. Арифметичні перетворення застосовуються для узгодження типів операндів у арифметичних і порозрядних операціях, а також в операціях порівняння. Вони базуються на "підтягуванні" операнда молодшого типу до старшого (цей процес ще називають *просуванням типів – type promotion*). У мові С встановлено таку ієрархію типів:

```
char < short ≤ int ≤ long < float < double < long double
```

Для цілих типів додатково діє умова:

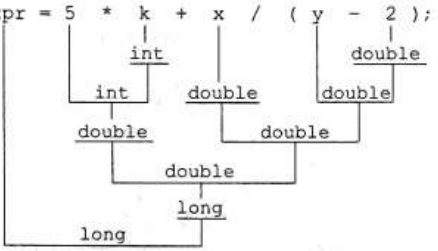
```
signed тип < unsigned тип
```

Підтягування типів виконується за наступною схемою:

1) всі операнди з типами `char` та `short` розширюються до типу `int`;
2) якщо старший операнд має один із цілочислових типів, а молодший є цілим зі знаком, то молодший операнд розширюється до розміру старшого через розмноження біта знака;
3) якщо старший операнд є цілим, а молодший – беззнакове число, то молодший операнд доповнюється до розміру старшого нульовими бітами;
4) перетворення `signed` *тип* у `unsigned` *тип* не змінює двійкового коду числа;
5) якщо старший операнд є дійсним, а молодший – цілим, то виконується перетворення "цілий → дійсний", яке реалізують спеціальні вбудовані засоби компілятора;
6) якщо обидва операнди є дійсними, то молодший розширюється до розміру старшого.

Розглянемо процес перетворення типів на прикладі оператора, що реалізує присвоєння значення арифметичного виразу змінній `expr`:

```
/*  Схема перетворення операндів у арифметичному виразі  */
    unsigned char k;
    float x;
    double y;
    long expr;
expr = 5  *  k  +  x  /  ( y  -  2 );
```



В операції `5*k` операнд `k` перетворюється до типу `int`, а пізніше, перед виконанням операції додавання, результат множення `5*k` з цілого типу перетворюється до типу `double`. Константа `2` у виразі `y-2` теж підтягується до типу `double`. Таким чином, весь вираз справа від знака `=` має тип `double`. У процесі виконання операції присвоєння значення арифметичного виразу перетворюється до типу змінної `expr`, тобто до довгого цілого типу `long`.

Особливу увагу слід звертати на вирази, в яких змішуються знакові та беззнакові операнди.

Проілюструємо це таким прикладом:

```
/*  Небезпечна помилка через перетворення int → unsigned  */
   int  c=-10;
   unsigned  d=10;
printf ("%s", c<=d ? "Так" : "Ні");
```

У результаті виконання функції `printf()` на екран буде виведено слово `Ні`, що вказує на хибність операції порівняння `c <= d`. Щоб пояснити причину такого дивного результату, наведемо внутрішні двійкові коди змінних `c` і `d`:

Пригадаємо, що від'ємні числа зберігаються в доповнювальному коді. Для виконання операції порівняння значення `c` підтягується до типу `unsigned int` без зміни свого двійкового коду. Порівняння двійкових кодів `c` і `d` як кодів двох беззнакових чисел виявляє, що код `c` є більшим за код `d`, а отже: `-10 > 10`!

Усунути помилку можна через перетворення обох операндів до старшого знакового типу. Вираз:

```
(long)c < (long)d ? "Так" : "Ні"
```

дасть правильний результат.

Перетворення типів в операціях присвоєння. В операціях присвоєння діє правило перетворення типу виразу, записаного справа від знака `=` до типу змінної чи *l*-операнда, яким присвоюється значення даного виразу. При цьому:

- якщо тип змінної (*l*-операнда) старший за тип виразу, значення якого присвоюється, то тип виразу підтягується до типу змінної за схемою, описаною вище;
- якщо тип змінної (*l*-операнда) молодший за тип виразу, то значення виразу перетворюється (обтинається) до типу змінної.

У наведеному раніше прикладі узгоджень типів у процесі обчислення значення змінної `expr` тип арифметичного виразу `double` перед виконанням операції присвоєння перетворюється до типу `expr`, тобто понижується до `long`.

Пониження типів пов'язане з такими перетвореннями:

1) перетворення числового значення старшого дійсного типу до молодшого дійсного за умови, що воно потрапляє в діапазон значень молодшого типу, реалізується як зменшення (через округлення) точності мантиси; інакше результат перетворення не визначений;
2) перетворення числового значення цілого типу до одного з молодших типів виконується через відкидання "зайвих" старших байтів числа; відкинуті старші байти втрачаються, що може призвести до втрати значущих старших розрядів числа, а для знакових типів додатково може відбутись зміна знака обрізаного числового значення,

пов'язана з тим, що в знаковий розряд потрапляє біт, значення якого інверсне до знака числа, яке присвоюється;

3) перетворення `unsigned` *тип* до `signed` *тип* не змінює розміру та двійкового коду числа, а найстарший біт `unsigned`-даного стає знаковим розрядом нового `signed`-значення; якщо цей біт дорівнює одиниці, то результат перетворення вважається від'ємним числом;
4) перетворення значення дійсного типу до цілого полягає у відкиданні (без округлення) дробової частини числа; якщо ціла частина числа перевищує межі, допустимі для цілих типів, то результат перетворення невизначений.

Присвоєння з пониженням типів потребують особливої уваги й обережності, щоб запобігти можливим помилковим результатам.

Наприклад, можна попередньо перевірити значення, яке присвоюється:

```
/*  Контроль присвоєння з пониженням типу  */
   double res;
   int zm;
res = . . .;                /*  обчислення double-значення змінної res */
if (res<=INT_MAX && res>=INT_MIN)        /* константи з <limits.h> */
   zm=res;                  /* значення res допустиме для присвоєння zm */
else
   puts ("Значення виходить за межі int");
```

Підкреслимо також, що в разі присвоєння дійсного значення цілочисловій змінній втрачається дробова частина числа. Тому два наступні присвоєння дадуть однаковий результат:

```
zm=2.1;  /* zm дорівнюватиме 2 */      zm=2.9;  /* zm дорівнюватиме 2 */
```

У виразах мови С часто використовують операцію явного перетворення типу (*тип*). Ми вже використовували цю операцію для отримання дійсної частки від ділення двох цілих операндів. Розглянемо ще один приклад:

```
   int u=212, w=197;
   double z;
z=u*w;
```

На перший погляд оператор присвоєння цілком коректний. Але, якщо вивести на екран значення `z`, то виявиться, що воно від'ємне (а саме: `-23772.0`). Причина в тому, що обидва операнди операції `u*w` мають тип `int`, отже, результат множення теж матиме тип `int`. Значення добутку (`41764`) перевищує величину `INT_MAX`, а його двійковий код є таким, що в біт знака потрапляє 1. Тому добуток інтерпретується як від'ємне число, яке перед присвоєнням перетворюється до типу `double`. Щоб уникнути помилки, треба хоча б один із операндів операції множення (краще обидва операнди) явно перетворити до дійсного типу:

```
z=(double)u*(double)w;
```

Наступний приклад є виразом, що повертає цілочислове значення, найближче до заданого дійсного x:

```
x > 0 ? (int)(x+0.5) : (int)(x-0.5)
```

Треба тільки пам'ятати, що значення x повинно потрапляти в діапазон допустимих значень типу int.

4.10. Стандартні математичні функції

У виразах обчислювального характеру часто доводиться застосовувати ті чи інші математичні функції: піднесення до степеня, логарифмування, знаходження кореня, обчислення тригонометричних значень тощо. Мова С підтримує бібліотеку основних математичних функцій, що дає змогу звертатись до них як до операндів арифметичних чи інших операцій.

Математичні функції С та відповідні макроси оголошені в стандартному заголовному файлі <math.h>, який у разі використання цих функцій треба підключити до програми директивою препроцесора:

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

Перелік найбільш уживаних математичних функцій наведено в табл. 4.3 (повний перелік функцій <math.h> бібліотеки Borland C подано в табл. Д2.1 Додатка 2). Всі математичні функції повертають дійсне значення, що має тип double. Якщо ж результат функції виходить за межі діапазону цього типу, то функція повертає значення макроконстанти HUGE_VAL. Аргумент x наведених математичних функцій може бути довільним арифметичним виразом зі значенням, що має тип double (інакше воно перетворюється до типу double). Значення x обов'язково повинно потрапляти в межі, встановлені для аргументів відповідної математичної функції.

Математичний вираз:

$$y = \sin 0{,}5x + \sqrt{3}\cos\frac{x}{x-1} - 2^{x-1}$$

у С-програмі можна так записати за допомогою стандартних бібліотечних функцій:

```
y = sin(0.5*x) + sqrt(3.0) * cos(x/(x-1)) - pow(2.0, x-1);
```

Звернемо увагу на функцію pow(x,y), яка виконує піднесення значення x до степеня y. Обидва її аргументи і результуюче значення дійсні, тому використовувати pow() для піднесення даних до цілого степеня недоцільно. Зокрема, для обчислення x^2 значно ефективнішим буде вираз x*x, ніж pow(x,2). У наступному параграфі показано, що для програмування операцій піднесення до цілого степеня можна ефективно використовувати відповідні макроси з параметрами.

Стандарт С-99 значно розширив математичну бібліотеку мови. По-перше, продубльовано більшість функцій та макросів для двох інших дійсних типів (float і long double), що підвищило мобільність С-програм. По-друге, бібліотеку доповнено рядом нових математичних функцій: обчислення кубічного кореня, гама-функції, гіпотенузи прямокутного трикутника тощо. Ознайомитись з нововведеннями можна в [24].

Основні математичні функції бібліотеки <math.h>

Функція	Зміст результату
`exp(x)`	e^x – експонента x
`log(x)`	$ln\,x$ – логарифм натуральний x (x > 0)
`log10(x)`	$lg\,x$ – логарифм десятковий x (x > 0)
`pow(x,y)`	x^y – піднесення x до дійсного степеня y (помилка, якщо: 1) x = 0, а y ≤ 0 ; 2) x < 0, а y – не є цілим)
`sqrt(x)`	$\sqrt{x}$ – корінь квадратний з x (x ≥ 0)
`fabs(x)`	$\|x\|$ – абсолютна величина x
`sin(x)`	$sin\,x$ – синус x (значення x задається в радіанах)
`cos(x)`	$cos\,x$ – косинус x (значення x задається в радіанах)
`tan(x)`	$tg\,x$ – тангенс x (значення x задається в радіанах)
`asin(x)`	$arcsin\,x$ – арксинус x у діапазоні $[-\pi/2, \pi/2]$ ($x \in [-1, 1]$)
`acos(x)`	$arccos\,x$ – арккосинус x у діапазоні $[0, \pi]$ ($x \in [-1, 1]$)
`atan(x)`	$arctg\,x$ – арктангенс x у діапазоні $[-\pi/2, \pi/2]$
`atan2(x, y)`	$arctg\,x/y$ – арктангенс x/y у діапазоні $[-\pi, \pi]$
`sinh(x)`	$sh\,x$ – гіперболічний синус x
`cosh(x)`	$ch\,x$ – гіперболічний косинус x
`tanh(x)`	$th\,x$ – гіперболічний тангенс x
`ceil(x)`	найменше ціле (у форматі `double`) ≥ x
`floor(x)`	найбільше ціле (у форматі `double`) ≤ x

4.11. Макроси з параметрами

Директива препроцесора `#define`, використання якої для макропідстановок константних значень розглядалось раніше, має ще одну цінну властивість: вона дає змогу створювати *макроси з параметрами* (їх ще називають *функціональними макровизначеннями*).

Загальний синтаксис макроса з параметрами такий:

`#define` *ім'я_макроса*`(`*список_параметрів*`)` *вираз_для_заміни*

Між іменем макроса і дужкою, за якою записується список параметрів, не повинно бути символів пробілу.

Макропідстановки з параметрами виконуються так: кожне звертання в програмі до заданого макроса препроцесор замінює виразом підстановки, в якому замість формаль-

них параметрів, вказаних у оголошенні макроса в рядку `#define`, записуються ті аргументи, що вказані в даному звертанні до нього.

Наведемо приклад. Нехай в програмі оголошено макрос:

```
#define  SQR(x)  (x)*(x)          /* макрос, що реалізує x2 */
```

Тоді вираз $a^2-2c^2+4(a-c)^2$ можна в цій програмі записати так:

```
SQR(a) - 2*SQR(c) + 4*SQR(a - c)
```

Звертаємо увагу, що операндами виразу є не виклики функції, а макропідстановки, тобто запис `SQR(a)` буде замінено виразом `(a)*(a)`, запис `SQR(c)` – виразом `(c)*(c)`, а останнє звертання `SQR(a-c)` замінюється макропідстановкою `(a-c)*(a-c)`. Якби в оголошенні макроса параметр `x` не був охоплений дужками, тобто якби директива була записана так:

```
#define  SQR(x)  x * x            /* макрос, що може викликати помилку */
```

то результати двох перших замін `a*a` та `c*c` були б коректними, а остання заміна дала би помилковий вираз: `a-c*a-c`. Щоб уникнути таких та інших можливих помилок, кожну складову частину виразу підстановки і весь вираз беруть у дужки:

```
#define  SQR(x)  ((x)*(x))        /* універсальний запис макроса */
```

Ось ще два приклади корисних макросів:

```
#define  ABS(x)  ((x)<0 ? -(x) : (x))          /* абсолютне значення */
#define  MAX(n1,n2)  ((n1)>(n2)?(n1):(n2))     /* більше з двох */
```

Використання макросів з параметрами замість викликів функцій дає певну перевагу в швидкодії роботи програми, оскільки підстановка здійснюється ще на етапі компіляції (препроцесування), тому в процесі виконання програми не витрачається час на звертання до функцій. Крім цього, аргументи, що вказуються у звертанні до макросів, можуть мати довільний тип (тип аргументів визначає тип результату виразу макропідстановки), тоді як типи параметрів і результату функції жорстко встановлюються в її оголошенні.

Якщо в програмі задекларовано:

```
unsigned int  m, n;
double  arg;
```

то результат звертання `MAX(m, 2*n)` матиме тип `unsigned int`, а результат другого звертання `MAX(-sin(arg/3), cos(arg))` – тип `double`.

Більше про макропідстановки з параметрами можна дізнатись у розділі 17.

Запитання та завдання для самоконтролю

1. З чого формують вирази мови С?
2. Які операції називають унарними, бінарними і тернарними? Наведіть приклади таких операцій.

3. Який порядок обчислення вказаних нижче виразів:
   1) `c+h/5-3*k`
   2) `x>100||x<=10&&y==0`
   3) `z>0?2/cos(z/2):z*z+0.5`
   4) `w+=u++*2*-k`
4. У програмі оголошено декілька цілочислових змінних:
```
int a=4,  b=2*a,  c='A';
unsigned  z1=0x0f2c,  z2=0xb17d;
```
   Знайдіть значення наступних виразів:
   1) `a+b/3-sizeof(a)`
   2) `a-1==(b+2)%a`
   3) `c+5>'0'`
   4) `z2>>b-5`
   5) `z1^!z2<=z2`
5. Як перетворюються операнди в бінарних арифметичних операціях, якщо ці операнди мають різний тип?
6. Як зміняться значення змінних `k`, `m` та `u` після виконання кожного зі записаних далі присвоєнь, якщо ці змінні оголошено так:
```
int  k=2,  m=5;
double  u=0.43;
```
   1) `k=m++;`
   2) `k=--m;`
   3) `k=m/2>3*u?m/2:3*u;`
   4) `u+=k+m;`
   5) `k=m*=(int)(5*u);`
7. Використовуючи стандартні бібліотечні математичні функції, запишіть мовою С такі вирази:
   1) $2\sin x+\cos^2 x/3$
   2) $\lg(x+\frac{a}{0,5x+1,8})$
   3) $\sqrt{7y^2z+|z^3|}$
   4) $\sqrt[3]{\frac{u+\mathrm{tg}ux^2}{\log_u 2x-u^3}}$
8. Задано дійсне число. Напишіть вираз, який буде істинним, якщо найближче ціле до даного числа є числом, що закінчується цифрою 7.
9. Задано довге ціле число. Напишіть вираз, який інвертує молодший байт цього числа (всі інші байти повинні залишитись незміненими).
10. Напишіть макрос з параметрами, значення якого дорівнюватиме 1, якщо з двох заданих значень меншим є перше, та 2, якщо меншим є друге значення.

Розділ 5

ФОРМАТНЕ ВИВЕДЕННЯ ТА ВВЕДЕННЯ ДАНИХ

У цьому розділі:

- Загальна характеристика стандартних бібліотечних функцій `printf()` та `scanf()`
- Параметри функції `printf()`, структура специфікацій формату
- Специфікації виведення: одиночних символів (`c`), символьних рядків (`s`), цілих десяткових чисел (`d` та `i`), цілих беззнакових чисел (`u`, `o`, `x/X`), дійсних чисел (`f`, `e/E`, `g/G`)
- Змінні рядки формату
- Функція `scanf()`: параметри функції, структура рядка формату та специфікацій формату
- Специфікації введення: одного символа (`c`), символьних рядків (`s`), цілих чисел (`d`, `u`, `o`, `x`, `i`), дійсних чисел (`f`, `e/E`, `g/G`)
- Введення символьних рядків за шаблоном, організація спеціальних форм введення

Для перевірки працездатності та правильності роботи програми, потрібно певним чином відобразити результати її виконання, а також організувати введення необхідних для виконання програми даних. Тому перед розглядом операторів мови С пропонуємо ознайомитись з бібліотечними функціями форматного виведення і введення даних, які з огляду на універсальність найчастіше використовуються для відображення на екрані інформаційних повідомлень програми і результатів її роботи та для введення з клавіатури значень вхідних даних.

Мова С не містить вбудованих (таких, що входять до складу компілятора) засобів, які б виконували введення/виведення даних. Всі операції обміну даними як з термінальними пристроями (дисплей, клавіатура тощо), так і з дисковими файлами, найчастіше реалізують через звертання до відповідних бібліотечних функцій. У цьому розділі розглянемо тільки дві з цих функцій: `printf()` та `scanf()`, з іншими будемо знайомитись пізніше, вивчаючи відповідні теми.

Функції `printf()` та `scanf()` призначені відповідно для виведення та введення даних різних типів, вони працюють з усіма стандартними скалярними типами, а також

з рядками символів, що завершуються нуль-символом. Обидві функції реалізують *форматний* обмін даними, тобто в процесі виведення чи введення даних відбувається їх перетворення (форматування) згідно зі заданими шаблонами формату.

Прототипи функцій `printf()` та `scanf()` оголошені в заголовному файлі <stdio.h>. Обидві функції належать до групи стандартних бібліотечних функцій т. зв. високорівневого потокоорієнтованого буферизованого обміну даними.

5.1. Форматне виведення даних

Форматне виведення заданої інформації на екран виконує функція:

```
int  printf (рядок_формату,  список виразів_виведення);
```

яка перетворює значення виразів, заданих у списку виведення, відповідно до специфікацій, записаних у рядку формату, і передає результат перетворення у стандартний вихідний потік `stdout`, який здебільшого пов'язаний з виведенням на екран. Функція повертає цілочислове значення. За умови успішного завершення це значення дорівнює загальній кількості виведених символів. У разі виникнення помилки в процесі виведення даних `printf()` повертає від'ємне значення.

Першим параметром функції `printf()` є *рядок_формату* – символьний рядок, до складу якого можуть входити три групи символів:

- звичайні ASCII-символи – вони виводяться без змін;
- керуючі символи (ескейп-послідовності) – виконують відповідну дію;
- *специфікації формату* – задають інтерпретацію і форму зображення відповідного за порядком параметра зі списку виведення; значення цього параметра буде підставлене замість специфікації формату в рядок виведення.

Якщо специфікації формату відсутні, то рядок формату виводиться як звичайний символьний рядок. Наприклад, якщо рядок формату `printf()` буде таким:

```
printf ("\t Почнемо! \n Введіть своє ім\'я - ");
```

то на екран виведеться наступний текст:

```
      Почнемо!
Введіть своє ім'я - _
```

тут символом _ позначено позицію екранного курсора після виконання `printf()`.

Специфікації формату функції `printf()` повинні бути попарно узгоджені в порядку зліва направо з виразами зі списку виведення, тобто кожній специфікації формату в списку виведення повинен відповідати один послідовний параметр. Параметри списку виведення можуть бути довільними виразами, але типи цих виразів обов'язково мають бути сумісними з відповідними специфікаціями формату, інакше виведене значення та всі наступні за ним будуть хибними.

Якщо параметрів виведення більше, ніж специфікацій у рядку формату, то останні зайві значення ігноруються (не виводяться і втрачаються). Якщо ж виразів у списку

замало, то результат виведення невизначений (найчастіше замість відсутніх даних виводиться т. зв. "сміття" – випадкові значення, якими на момент виведення заповнено відповідну ділянку оперативної пам'яті).

Структура специфікацій формату. Кожна специфікація формату починається знаком `%` і закінчується однією із літер-специфікаторів. Загальна структура специфікацій формату функції `printf()` така:

`%` [*прапорці*][*ширина*][. *точність*][*модифікатор*] *літера-специфікатор*

Квадратними дужками [ ] позначені ті елементи специфікацій, які не є обов'язковими; їх використовують, щоб розширити базові властивості специфікацій. Обов'язковими для специфікацій є тільки два елементи: символ `%` та літера-специфікатор.

Усередині специфікації формату не повинно бути символів пробілу (крім спеціального прапорця, про який мова йтиме далі).

Призначення елементів специфікації формату наступне:

- `%` – ознака початку специфікації (кінцем слугує літера-специфікатор);
- *прапорці* – керують формою зображення і розташуванням даного. Набір допустимих прапорців залежить від конкретної специфікації формату;
- *ширина* – ціле число, що встановлює ширину поля виведення, тобто задає мінімальну кількість позицій у рядку виведення, які буде займати зображення відповідного параметра. Якщо цих позицій недостатньо, то елемент *ширина* ігнорується, а розмір поля встановлюється згідно зі значенням параметра і характеристиками даної специфікації; якщо ж *ширина* завелика, то зайві позиції заповнюються символом пробілу. З елементом *ширина* пов'язаний прапорець *мінус* (позначається знаком -), який записують відразу після знака `%`. Наявність цього прапорця вказує, що значення, яке виводиться, має бути притиснутим до лівого краю поля виведення, а зайві праві позиції заповнюються пробілами, інакше (якщо прапорець - не вказано) виведене значення притискається до правого краю, а пробіли виводяться перед ним. Ширину поля виведення і прапорець вирівнювання можна задавати для всіх специфікацій формату;
- *точність* – починається символом ., за яким вказується ціле число, що визначає кількість цифр або символів у значенні, яке виводиться. Конкретні властивості параметра *точність* залежать від специфікації формату;
- *модифікатор* – конкретизує тип числового аргументу;
- *літера-специфікатор* – встановлює, як буде інтерпретуватись відповідний параметр списку виведення, і задає форму його відображення.

Оскільки знак `%` є ознакою початку специфікації формату, то, щоб вивести в повідомленні символ `%`, треба записати його двічі (продублювати) – `%%`:

```
printf("\t Виконано %d%% усієї роботи \n", result);
```

Якщо значення змінної `result` дорівнює 15, то на екран буде виведено:

```
Виконано 15% усієї роботи
```

Далі розглянемо особливості застосування основних специфікацій формату.

Виведення одиночних символів (специфікація c). Для відображення одиночних символів застосовують специфікацію, яка має таку структуру:

%[-][*ширина*]c

(як і раніше, будемо відзначати квадратними дужками ті елементи специфікацій, що не належать до обов'язкових).

Специфікація c вказує, що на екран буде виведено символ, код якого (ASCII-код) задає відповідний параметр із списку виведення. Цей параметр може бути довільним виразом, що має тип char або int, обидва типи можуть бути як знаковими, так і беззнаковими. У разі типу int до уваги береться тільки значення молодшого байта даного.

Подамо приклад виведення зображення та коду символа.

```
int sym=65;
printf("Символ - %c, код - %d.\n", sym, sym);
```

У процесі виконання printf() на екран буде виведено:

```
Символ - A, код - 65.
```

Додаткові елементи специфікації формату [-] та [*ширина*] керують розміщенням символа. Прапорець *мінус* вказує, де буде відображено символ: з лівого чи з правого краю поля виведення, розмір якого задає цілочислове значення елемента *ширина*. Проілюструємо дію цих елементів наступним прикладом трьох варіантів виведення символа:

```
printf("\nВаріант1=>%c", '*');
printf("\nВаріант2=>%3c", '*');
printf("\nВаріант3=>%-3c", '*');
```

Результатом виконання цих операторів будуть три екранні повідомлення:

```
Варіант1=>*_
Варіант2=>  *_
Варіант3=>*  _
```

тут символом _ позначено позицію курсора після виконання відповідної функції.

Виведення символьних рядків (специфікація s). Для виведення на екран цілого рядка символів або його частини застосовують наступну специфікацію:

%[-][*ширина*][.*точність*]s

Перші два необов'язкові елементи задають форму вирівнювання і ширину поля виведення (якщо вона недостатня, то елемент *ширина* ігнорується). Елемент *точність* задає кількість початкових символів рядка, що мають бути виведені (якщо він більший за розмір рядка, то виведення припиняється, коли досягнуто кінця рядка).

Відповідний параметр у списку виведення повинен бути рядком символів, що закінчується нуль-символом '\0'. Це може бути константний рядок або вказівник на перший символ рядка, який треба вивести.

Наведемо кілька прикладів. Нехай у програмі оголошено:

```
char sent[]= "Мова програмування С";
```

і записано наступні оператори виклику функції `printf()`:

```
printf("  1. %s \n", sent);
printf("  2. %8.4s \n", sent);
printf("  3. %1.4s \n", sent);
printf("  4. %-15.13", sent+5);
```

На екран буде виведено:

```
1. Мова програмування С
2.     Мова
3. Мова
4. програмування _
```

тут символ _ вказує позицію курсора після виконання всіх функцій `printf()`.

Виведення цілих десяткових чисел (специфікації `d` та `i`). Для виведення цілих чисел у десятковій формі зі знаком застосовують специфікації формату з літерами `d` або `i`. Відповідний параметр у списку виведення `printf()` може бути довільним цілочисловим виразом. Значення цього виразу буде розглядатись як ціле число зі знаком. Дія специфікацій `d` та `i` у разі виведення даних однакова, відмінність між ними проявляється тільки у функції `scanf()`.

Повна форма специфікації виведення цілого десяткового числа така:

`%[-][+/_][0][`*ширина*`][.`*точність*`][h/l]d/i`

тут коса риска / відокремлює альтернативні варіанти, з яких у специфікації можна використати один, а знаком підкреслення _ позначено пробіл. Тобто запис [+/_] означає, що в даному місці специфікації може бути записаний знак + або символ пробілу.

Призначення необов'язкових елементів специфікацій `d` та `i` наступне:

- прапорець - (мінус) – вказує, що число буде вирівнюватись на лівий край поля виведення, заданого елементом *ширина*;
- прапорець +/_ – задає форму виведення знака додатних чисел. Стандартно від'ємні числа виводяться зі знаком мінус, а додатні – без ніякого знака. Прапорець + вказує, що відповідне додатне число буде виведене зі знаком плюс, а прапорець _ (пробіл) встановлює, що перед додатним числом буде стояти символ пробілу. Ось приклад:

```
int n=25;
printf ("n(1)=>%d    n(2)=>%+d    n(3)=>% d \n", n, n, n);
```

Результат виведення виглядатиме так:

```
n(1)=>25    n(2)=>+25    n(3)=> 25
```

- прапорець 0 – забезпечує доповнення числа спереду символом 0 до заданої точності або ширини поля, якщо точність не вказано. Наприклад:

```
printf ("n(4)=>%06d    n(5)=>%06.4d \n", n, n);
```

Результат виведення:

```
n(4)=> 000025      n(5)=>  0025
```

- *ширина* – задає мінімальний розмір поля виведення;
- *точність* – задає мінімальну кількість цифр у числі, що виводиться. Якщо ця кількість недостатня або відсутній прапорець `0`, то значення точності до уваги не береться, інакше число доповнюється необхідною кількістю початкових нулів, як у випадку `n(5)`;
- модифікатор `h` / `l` – конкретизує цілочисловий тип.

Якщо відповідне дане зі списку виведення має тип `long`, то слід обов'язково вказувати модифікатор `l`. Модифікатор `h` застосовують для виведення даних з типом `short`.

За відсутності модифікатора типу виведене значення й усі наступні за ним можуть бути хибними.

Нехай у програмі оголошено:

```
long int a=65600;
short int b=108;
```

Якщо виведення значень цих змінних організувати так:

```
printf("\t a=%ld    b=%hd \n", a, b);
```

то на екрані будуть відображені правильні значення:

```
a=65600    b=108
```

Якщо ж модифікатори типів опустити:

```
printf("\t a=%d    b=%d \n", a, b);
```

то результат виведення може бути, приміром, таким:

```
a=64    b=1
```

Виведення цілих беззнакових чисел (специфікації `u`, `o`, `x`, `X`). Дані специфікації формату застосовують для виведення цілих чисел, що інтерпретуються як беззнакові. Так само, як і для специфікацій `d` та `i`, відповідний параметр зі списку виведення повинен бути виразом цілого типу, але значення цього виразу розглядається як беззнакове число.

Три різні літери, що використовуються у специфікаціях беззнакових чисел, позначають три форми відображення числа:

- `u` (від слова *unsigned*) – значення числа виводиться в десятковій формі;
- `o` (*octimal* – вісімковий) – значення числа виводиться у вісімковій формі;
- `x` та `X` (*heximal* – шістнадцятковий) – значення числа виводиться у шістнадцятковій формі з використанням малих (`x`) або великих (`X`) латинських літер.

Розглянемо приклад. Нехай у програмі оголошено беззнакову змінну `w`:

```
unsigned int w=1000;
```

Тоді результатом звертання до функції `printf()`:

```
printf ("w(u)=> %u     w(o)=> %o     w(x)=> %x     w(X)=> %X \n",
          w, w, w, w);
```

буде рядок значень змінної w, виведених за вказаними специфікаціями формату:

```
w(u)=> 1000     w(o)=> 1750     w(x)=> 3e8     w(X)=> 3E8
```

Повна форма специфікації виведення цілого беззнакового числа така:

%[-][#][0][*ширина*][.*точність*][h / l]*літера-специфікатор*

тут *літера-специфікатор* – це одна із літер: u, o, x або X.

Призначення елементів специфікацій [-], [0], [*ширина*], [.*точність*], [h/l] таке ж, як і для специфікацій d та i. Прапорець знака у специфікаціях беззнакових чисел відсутній, але вони мають власний прапорець – #, який керує відображенням префікса числа в специфікаціях o, x та X. Наявність прапорця # вказує, що відповідне вісімкове значення буде виводитись із префіксом 0, а шістнадцяткове – з префіксом 0x (специфікація x) або 0X (специфікація X).

Приклад виведення чисел у вісімковій і шістнадцятковій формі з префіксом:

```
printf ("w(#o)=> %#o    w(#x)=> %#x    w(#X)=> %#X \n", w, w, w);
```

Результат виведення:

```
w(#o)=> 01750     w(#x)=> 0x3e8     w(#X)=> 0X3E8
```

Префікси вісімкових і шістнадцяткових чисел входять у загальну кількість цифр цього числа, їх треба враховувати, коли задається ширина поля виведення та точність числа. Це ілюструють наступні приклади:

```
printf ("w(1)=> %04x    w(2)=> %#06x    w(3)=> %#10.8X \n", w, w, w);
printf ("w(4)=> %04o    w(5)=> %#06o    w(6)=> %#10.8o \n", w, w, w);
```

Результат виведення:

```
w(1)=> 03e8     w(2)=> 0x03e8     w(3)=>  0X0003E8
w(4)=> 1750     w(5)=> 001750     w(6)=>  00001750
```

У разі, якщо дане, значення якого виводиться через специфікацію u, o, x або X має знаковий тип, а значення його від'ємне, то це значення буде інтерпретуватись як двійковий код відповідного беззнакового числа.

Наведемо приклад виведення від'ємного значення змінної g.

```
 int g=-5;
printf ("g(d)=> %d     g(u)=> %u     g(o)=> %o     g(x)=> %x\n",
          g, g, g, g);
```

Для систем програмування, в яких від'ємні числа зберігаються у доповнювальному коді, результат виведення буде таким:

```
g(d)=> -5     g(u)=> 65531     g(o)=> 177773     g(x)=> fffb
```

Виведення дійсних значень (специфікації `f`, `e`, `E`, `g`, `G`). Ці специфікації застосовують для відображення дійсних чисел – відповідний параметр списку виведення `printf()` повинен мати тип `float`, `double` або `long double` (останній вимагає модифікатора `L`).

Літера-специфікатор встановлює форму зображення числа:

- `f` – значення виводиться у формі числа з фіксованою крапкою, тобто вказуються ціла і дробова частини числа, розділені десятковою крапкою (стандартно дробова частина містить шість цифр);
- `e`, `E` – значення виводиться у формі числа з плаваючою крапкою, тобто через мантису і порядок. Мантиса записується як число з фіксованою крапкою, що має одноцифрову цілу частину, а порядок – як ціле двоцифрове (або трицифрове) число зі знаком. Мантиса відокремлюється від порядку літерою `e` або `E` залежно від літери-специфікатора;
- `g`, `G` – цей формат передбачає автоматичний вибір форми виведення числа: якщо задана точність (стандартно вона дорівнює шести) дає змогу вивести значення параметра в формі числа з фіксованою крапкою, то застосовується ця форма, а загальна кількість цифр у виведеному числі дорівнює точності; інакше значення параметра виводиться у формі числа з плаваючою крапкою.

Подамо приклад виведення дійсного значення змінної `z` через різні специфікації:

```
double z=1234.56789;
printf(" z(f)=>%f \n z(e)=>%e     z(E)=>%E \n z(g)=>%g\n",
          z, z, z, z);
```

Результат відобразиться на екрані у наступній формі:

```
z(f)=>1234.567890
z(e)=>1.234568e+03     z(E)=>1.234568E+03
z(g)=>1234.57
```

Повна форма специфікацій виведення дійсних чисел така:

`%[-][+/_][#][0][`*ширина*`][.`*точність*`][l/L]`*літера-специфікатор*

тут *літера-специфікатор* – це одна із літер: `f`, `e`, `E`, `g`, `G`.

Призначення прапорців та інших необов'язкових елементів специфікацій:

- прапорець `-` (мінус) – вирівнює число на лівий край поля виведення;
- прапорець `+/_(_` позначає символ пробілу) – встановлює форму виведення знака додатних чисел (його роль така сама, як і у випадку специфікацій цілих чисел);
- прапорець `#` – гарантує виведення десяткової крапки навіть у випадку, коли дробова частина числа відсутня (задано точність `0` для специфікацій `f` або `e/E`), а для специфікації `g/G` додатково забезпечує виведення кінцевих нулів у дробовій частині числа, щоб сумарна кількість цифр числа дорівнювала заданій точності (стандартно незначущі нулі дробової частини числа в специфікації `g/G` не виводяться);
- прапорець `0` – доповнює число спереду символами `0` до заданої ширини поля;
- *ширина* – задає мінімальний розмір поля виведення;

- *точність* – для специфікацій `f` та `e/E` задає кількість цифр у дробовій частині числа (виконується округлення), а для специфікацій `g/G` встановлює загальну кількість цифр числа (якщо точність не дає змогу вивести число з фіксованою крапкою, то воно виводиться у формі числа з плаваючою крапкою);
- модифікатор `l/L` – літеру-модифікатор `l` часто використовують для виведення даних, що мають тип `double`, хоча потреби в цьому модифікаторі немає (він необхідний тільки у разі введення даних); модифікатор `L` є обов'язковим, коли виводиться значення з типом `long double`.

Наведемо приклади, що ілюструють різні форми виведення дійсних значень.

1. Керування точністю і формою виведення знака змінної `z`, оголошеної вище (зверніть увагу на зображення чисел за специфікацією `g`):

```
printf("z(1)=>%1.3f   z(2)=>%+.1f   z(3)=>%09.2f \n", z, z, z);
printf("z(4)=>%.3e   z(5)=>%-14.4e   z(6)=>% 1.2e\n", z, z, z);
printf("z(7)=>%1.5g   z(8)=>%+1.4g   z(9)=>%.3g \n", z, z, z);
```

Результат виведення:

```
z(1)=>1234.568    z(2)=>+1234.6    z(3)=>001234.57
z(4)=>1.235e+03    z(5)=>1.2346e+03    z(6)=> 1.23e+03
z(7)=>1234.6    z(8)=>+1235    z(9)=>1.23e+03
```

2. Демонстрація використання прапорця `#`:

```
double r=200;
printf("r(1)=>%f   r(2)=>%1.0f   r(3)=>%#1.0f \n", r, r, r);
printf("r(4)=>%g   r(5)=>%#g   r(6)=>%.2g   r(7)=>%#.2G \n",
       r, r, r, r);
```

Результат виведення:

```
r(1)=>200.000000    r(2)=>200    r(3)=>200.
r(4)=>200    r(5)=>200.000    r(6)=>2e+02    r(7)=>2.0E+02
```

3. Виведення даного з типом `long double`:

```
long double y=-0.0608;
printf("y(1)=>%Lf   y(2)=>%1.1Lf   y(3)=>%11.2LE \n", y, y, y);
```

Результат виведення:

```
y(1)=>-0.060800    y(2)=>-0.1    y(3)=>  -6.08E-02
```

Змінний рядок формату. В усіх попередніх прикладах рядок формату задавався як стрінгова константа безпосередньо в функціях `printf()` та `scanf()`. Можна також задавати параметр рядка формату через відповідну змінну-вказівник, а сам рядок оголошувати та заповнювати окремо.

У наступному прикладі використано рядок формату, оголошений зовні:

```
char format[]="v1=%-12.2f  v2=%-12.2f  v3=%-12.2f\n";
double x1, x2, x3;
```

```
printf (format, x1*x2, x1*x3, x2*x3);
printf (format, x1/(x2+x3), x1/(x2-x3), x1/(x2*x3));
```

Такий підхід дає змогу вводити рядок формату з клавіатури, формувати його в процесі виконання програми, змінювати окремі специфікації тощо.

Іншою корисною властивістю функції `printf()` є те, що значення ширини поля і точності в специфікації виведення можуть бути змінними. Тоді в рядку формату замість констант ширини і точності вказують символ `*`, а відповідні значення зчитуються зі списку параметрів виведення.

Приклад зовнішнього задання ширини поля і точності для дійсного параметра:

```
printf (" Число -> %-*.*f " 8, 3, 56.78012);
```

За даними цього прикладу для специфікації `f` буде встановлено ширину поля виведення `8` і точність `3`. Результат виведення виглядатиме так:

```
Число -> 56.780
```

Безумовно, задавати константи в списку виведення недоцільно, це зроблено тільки для ілюстрації порядку запису та взаємозв'язку параметрів `printf()`. Якщо ж використати змінні значення ширини й точності, то можна забезпечити високу гнучкість виведення та легко пристосувати формати до вимог конкретних задач.

5.2. Форматне введення даних

Форматне введення даних зі стандартного вхідного потоку `stdin` (переважно він пов'язаний з клавіатурою) виконує функція `scanf()`:

`int scanf (` *рядок_формату, список адрес_введення* `);`

Ця функція послідовно читає введені символи, формуючи з них поля введення (вхідні дані). Кожне поле інтерпретується і перетворюється у двійкову форму відповідно до заданих у рядку формату специфікацій і внутрішніх кодів, властивих даній системі програмування. У разі успішного перетворення отримані значення записуються в змінні (ділянки оперативної пам'яті), адреси яких вказані у списку введення. Функція `scanf()` забезпечує введення даних усіх числових типів (як цілих, так і дійсних), а також одиночних символів і символьних рядків.

Рядок_формату, що є обов'язковим першим параметром функції `scanf()`, складається зі специфікацій, які визначають спосіб інтерпретації введених даних. Здебільшого *рядок_формату* записується як константний рядок, але його можна задавати також і через зовнішній символьний рядок. Для застосування зовнішнього рядка формату треба вказати в `scanf()` ім'я змінної, яка зберігає адресу цього рядка, подібно до того, як це було зроблено в наведеному вище прикладі для функції `printf()`.

Список введення `scanf()` складається з адрес змінних, які повинні отримати значення у процесі виконання функції, – підкреслимо, що значеннями параметрів функції мають бути саме ***адреси***. У загальному випадку список введення `scanf()` формується із вказівників, кожен з яких задає адресу ділянки оперативної пам'яті,

в яку буде записано введене значення. Кількість параметрів у списку введення повинна строго відповідати кількості специфікацій формату, а типи змінних, в які записуються введені значення, обов'язково повинні бути сумісними з відповідними специфікаціями перетворення даних.

Якщо параметрів введення більше, ніж специфікацій у рядку формату, то останні зайві параметри ігноруються (їх значення не вводяться). Якщо ж адрес у списку введення недостатньо, то результат роботи `scanf()` стандартом С не обумовлений (він залежить від конкретної реалізації функції). В обох випадках невідповідність кількості параметрів може призвести до помилок у роботі програми.

Як вже зазначалось, `scanf()` належить до функцій, які здійснюють потокоорієнтоване буферизоване введення даних. Вхідні дані заносяться в буфер введення і надходять на опрацювання в `scanf()` тільки після того, як натиснено клавішу *Enter*. До цього моменту їх можна виправляти, стирати чи редагувати.

Поля введення. Значення, яке повертає `scanf()`. Послідовність символів, записану в буфер введення, функція `scanf()` розбиває на поля, кожне з яких відповідає одному даному. Якщо у відповідній специфікації формату не задано ширину поля, то кінцем поточного поля слугує перший пробільний символ: пробіл, табуляція чи символ нового рядка. Всі записані підряд пробільні символи з вхідного потоку розглядаються як один роздільник полів, перший непробільний символ вважається початком нового поля введення (кома, крапка з комою та інші розділові знаки не є роздільниками полів і опрацьовуються як усі інші непробільні символи). Кінцем поточного поля також вважається перший помилковий символ, який не належить до групи символів, що використовуються для запису даних відповідного типу (наприклад, крапка, літера чи розділовий знак у разі введення цілого десяткового числа). Цей символ не зчитується і розглядається як початок наступного поля введення.

У процесі зчитування даних `scanf()` пропускає роздільники і опрацьовує поля введення. Кожне поле введення інтерпретується як окреме значення. Це значення перетворюється згідно з вказаною специфікацією у внутрішню двійкову форму, яка властива даним відповідного типу. В разі успішного перетворення отримане двійкове значення заноситься в оперативну пам'ять за адресою, заданою у списку введення. Якщо ж перетворити поле не вдається, то фіксується помилка і введення даних припиняється, тобто поточне поле і всі наступні поля залишаються в буфері введення, а відповідний параметр із списку введення і всі наступні за ним будуть незаповненими (значеннями відповідних змінних буде т. зв. "сміття").

Функція `scanf()` завершує роботу, коли кількість введених і опрацьованих полів дорівнює кількості специфікацій введення у рядку формату, або, коли виявлено помилкове заповнення поля введення. У цих випадках `scanf()` повертає кількість успішно введених і записаних в оперативну пам'ять значень (`0`, якщо не введено ні одного значення). Якщо ж перед введенням першого поля зустрінеться символ "кінець файла" (на клавіатурі він набирається комбінацією клавіш *Ctrl+z*), то функція `scanf()` повертає значення стандартної макроконстанти `EOF` (`EOF` оголошено в `<stdio.h>`, у більшості систем програмування вона дорівнює `-1`).

Доцільно в програмі контролювати значення, яке повертає функція `scanf()`, щоб оперативно виявляти випадки неправильного введення даних, зокрема недозаповнення списку введення через певні помилки в записі вхідних даних. Інакше виконання програми буде продовжуватись із невизначеними чи помилковими значеннями змінних.

Наведена нижче програма реалізує цикл введення довільної кількості цілих чисел через перевірку значення, яке повертає функція `scanf()`. Введення завершується, коли замість наступного числа користувач вводить якусь літеру або інший нецифровий символ. Оскільки в рядку формату вказано специфікацію `%d`, яка інтерпретує кожне вхідне поле як ціле десяткове число, то в разі введення нецифрового символа фіксується помилка і `scanf()` повертає значення `0` (тобто число не введено), що є ознакою закінчення процесу введення даних і нагромадження їх суми.

```
/*****************************************************************/
/*  Обчислення середнього значення послідовності введених чисел  */
/*****************************************************************/
#include <stdio.h>
void main (void)
{
    int  n=0,                    /* кількість введених чисел */
         numb;                   /* значення числа */
    long  sum=0 ;                /* сума чисел */
  printf("Вводьте числа, кінець введення - довільна літера:\n");
  while (scanf("%d", &numb)==1) {
     sum+=numb ;
     n++;
  }
  printf("Середнє значення %d чисел - %.2f\n", n, (double)sum/n);
}
```

Приклад виконання:

```
Вводьте числа, кінець введення - довільна літера:
17  29  32  57  72  103  127  q
Середнє значення 7 чисел - 62.29
```

Якщо кількість даних, введених з клавіатури, перевищує кількість специфікацій у рядку формату функції `scanf()`, то зайві дані не зчитуються і залишаються в буфері введення. Крім цього, після введення числових даних у буфері залишається код клавіші *Enter*, якою завершують кожне введення даних, тобто символ `'\n'` (чи `'\x0a'` в шістнадцятковому позначенні). Наступний виклик `scanf()` або іншої функції буферизованого введення розпочне роботу зі зчитування даних, які залишились у буфері.

Щоб очистити буфер введення, можна скористатись спеціальною бібліотечною функцією `fflush()` (її оголошено в заголовному файлі `<stdio.h>`), яка звільняє буфер стандартного потоку введення `stdin` від усіх незчитаних даних і символів:

```
fflush(stdin);
```

Структура рядка формату та специфікацій функції `scanf()`. Рядок формату `scanf()` може включати символи трьох видів:

- специфікації перетворення даних, які починаються знаком `%` і закінчуються однією із літер-специфікаторів, – задають спосіб інтерпретації поточного поля введення;
- пробільні символи (пробіл, горизонтальна чи вертикальна табуляція) – вказують, що треба пропустити всю послідовність роздільників, які записані в даному місці вхідного потоку, і перейти на перший символ наступного поля даних; оскільки пропуск роздільників для числових даних і символьних рядків виконується автоматично, то розділяти специфікації цих даних пробільними символами недоцільно; пробільні символи є істотними тільки у разі введення одного символа через специфікацію `%c`;
- інші символи – у вхідному потоці в даному місці повинен бути саме такий символ, він зчитується, але нікуди не заноситься; якщо ж відповідний символ у потоці даних відсутній, то фіксується помилка введення.

Загальна форма запису специфікацій форматного рядка `scanf()` наступна:

`%` [`*`][*ширина*][*модифікатор*] *літера-специфікатор*

Три елементи, записані в квадратних дужках, не обов'язкові.

Призначення необов'язкових елементів специфікацій таке:

- прапорець `*` – вказує, що поточне поле введення треба пропустити: значення поля зчитується відповідно до заданої специфікації формату, але воно не перетворюється і нікуди не записується (приклади використання прапорця `*` наведено далі);
- *ширина* – ціле число, що встановлює максимально можливий розмір поля введення (у символах); якщо у межах даного поля зустрінеться пробільний або помилковий символ, то відразу фіксується кінець поля;
- *модифікатор* – конкретизує тип числових специфікаторів, обов'язково має бути вказаний у разі введення даних, що мають тип `long`, `double` або `long double`.

Далі розглянемо детальніше дії окремих специфікацій введення функції `scanf()`.

Введення одного символа (специфікація `c`). Ця специфікація призначена для зчитування поточного символа з вхідного потоку, навіть якщо це пробільний символ. Відповідний параметр у списку введення повинен бути адресою даного з типом `char`.

Приклад, що ілюструє введення трьох послідовних символів:

```
 char  c1, c2, c3;
scanf("%c%c%c", &c1, &c2, &c3);
```

Якщо з клавіатури введено рядок

```
A_52
```

де символом _ позначено пробіл, то в змінну `c1` буде записано код символа `'A'`, в змінну `c2` – код символа пробілу `'_'`, а в `c3` – код символа `'5'`. Символ `'2'` та символ `'\n'` (від клавіші *Enter*) залишаться у буфері введення.

Якщо в рядку формату перед специфікаціями `%c` записати символи пробілу:

```
scanf(" %c %c %c", &c1, &c2, &c3);
```

і ввести рядок

```
____A_52
```

то перед введенням кожного із символів будуть пропущені всі роздільники, тобто `c1` отримає значення `'A'`, `c2 – '5'`, а `c3 – '2'`.

Для введення одного символа частіше використовують інші стандартні бібліотечні функції, насамперед `getchar()`, яку описано в розділі 9.

Введення символьного рядка (специфікація s). Якщо потрібно ввести з клавіатури групу послідовних символів, то застосовують специфікацію `s`. Зчитані символи записуються за заданою адресою, в кінець рядка завжди заноситься нуль-символ. Введення починається з першого непробільного символа і завершується, коли зустрінеться роздільник полів або коли зчитано кількість символів, яка задана шириною поля.

Специфікація `%s` не дає змоги вводити рядки, що містять символи пробілу.

Відповідний параметр у списку введення повинен задавати адресу, за якою буде записано введений рядок. Найчастіше він є іменем масиву символів, оскільки в мові С ім'я масиву зберігає адресу його першого елемента.

Дуже важливо, щоб параметр введення задавав адресу ділянки, обсяг якої є достатнім для розташування всіх введених символів, включаючи `'\0'` (компілятор С не організовує автоматичного контролю перевищення розмірів масиву).

Наведемо приклад введення символьного рядка:

```
  char sent[100];
scanf("%s", sent);
```

У разі введення з клавіатури наступного рядка:

```
    Контроль вхідних даних
```

у масив `sent` буде записано слово `"Контроль"` з нуль-символом у кінці (початкові символи пробілу й табуляції відкидаються). Решта рядка, починаючи із наступного за словом пробілу, залишиться у буфері введення.

Результат введення буде таким самим, якщо в `scanf()` використати специфікацію `%30s`. Хоча ця специфікація встановлює ширину поля, що дорівнює 30 символам, але в даному полі є символи пробілу, тому введений рядок буде обмежений першим словом речення. Якщо ж ширина поля дорівнюватиме 5:

```
scanf("%5s", sent);
```

то в разі введення такого ж рядка у масив `sent` буде записано послідовність символів `"Контр"` з кінцевим `'\0'`.

Специфікацію `%1s` використовують для зчитування з вхідного потоку першого непробільного символа.

Введення символьного рядка, в якому є символи пробілу, можна виконати, зокрема, за допомогою бібліотечної функції `gets()`, яку розглядатимемо в розділі 9.

Введення цілих чисел (специфікації d, u, o, x, i). Перелічені специфікації призначені для зчитування цілочислових даних. Літера-специфікатор вказує, як має бути записане число, яке вводиться:

- `d` – число повинно бути десятковим, перед числом може стояти знак `+` або `-`;
- `u` – число повинно бути десятковим без знака;
- `o` – число повинно бути вісімковим, префікс `0` вказувати не обов'язково;
- `x` – число повинно бути шістнадцятковим, префікс `0x` вказувати не обов'язково;
- `i` – число може бути десятковим, вісімковим чи шістнадцятковим, але для вісімкових і шістнадцяткових чисел обов'язковими є префікси `0` та `0x`.

Відповідна адреса в списку введення функції `scanf()` має бути адресою змінної з типом `int` або `unsigned int`.

Наведемо приклади введення цілих чисел. Нехай в програмі оголошено змінні:

```
int a, b;
unsigned c;
```

Якщо введення значень цих змінних виконує наступна функція:

```
scanf("%d%d%x", &a, &b, &c);
```

і з клавіатури введено цілочислові дані:

```
+25  -103  25
```

то змінна `a` отримає значення `25`, змінна `b` – значення `-103`, а змінна `c` – значення `37` (число `25` вводиться за специфікацією `%x`, тому інтерпретується як шістнадцяткове).

Якщо для введення даних використати специфікацію `%i`:

```
scanf("%i%i%i", &a, &b, &c);
```

і ввести наступні числа:

```
0x10  010  10
```

то значення змінних будуть такими: `a – 16`, `b – 8`, `c – 10` (перше число розглядається як шістнадцяткове, друге – як вісімкове, а останнє – як десяткове).

Якщо дані записані неперервним потоком (без роздільників), то в специфікаціях формату треба вказувати розмір кожного поля введення (крім останнього), наприклад:

```
scanf("%3d%*2d%4d%u", &a, &b, &c);
```

У разі введення числового рядка

```
12233456789
```

змінні отримають такі значення: `a – 122`, `b – 4567`, `c – 89`. Специфікація `%*2d` задає пропуск другого поля, тобто значення `33`.

Для введення значень змінних, що мають тип `long int` та `unsigned long`, треба обов'язково вказувати модифікатор `l`, а для даних з типом `short int` та `unsigned short` – модифікатор `h`.

Наступний приклад ілюструє введення даних з типом `long` та `short`:

```
long int  g;
unsigned short  h;
scanf("%ld%hu", &g, &h);
```

Введення дійсних чисел (специфікації `f`, `e`, `E`, `g`, `G`). У функції `scanf()` дія всіх перелічених специфікацій щодо введення дійсних чисел збігається. Вхідні дані можуть бути записані у формі цілих чи дійсних чисел з фіксованою або з плаваючою крапкою. Для чисел з плаваючою крапкою мантиса може бути цілою або дійсною (число з фіксованою крапкою), а порядок повинен бути цілим зі знаком або без знака; мантиса відокремлюється від порядка літерою `e` або `E`.

Незалежно від форми запису вхідного числа у процесі введення воно перетворюється у внутрішню форму даного з типом `float`. Тому відповідний параметр у списку введення повинен бути адресою змінної (ділянки пам'яті), що має тип `float`.

Якщо введене число треба перетворити в тип `double`, то перед літерою-специфікатором слід обов'язково вказати модифікатор `l`, а для перетворення у тип `long double` – модифікатор `L`.

Подамо приклад зчитування трьох дійсних чисел і виведення їх значень .

```
float  r;
double  x, z;
scanf("%f%lf%lf", &r, &x, &z);
printf("r = %.2e     x = %.2e     z = %.2e \n", r, x, z);
```

Якщо дані введено з клавіатури так:

```
450  0.00781  165.8e-12
```

то на екрані відобразиться наступний результат:

```
r = 4.50e+02     x = 7.81e-03     z = 1.66e-10
```

Ще раз звернемо увагу, що для введення значення даного з типом `double` необхідно застосовувати модифікатор `l` (інакше внутрішній двійковий код відповідного числа буде помилковим, бо відповідатиме формату `float`), а в разі виведення даного з типом `double` модифікатор `l` не є обов'язковим, оскільки перед виведенням всі дані з типом `float` автоматично перетворюються в тип `double`. Для даних, що мають тип `long double`, модифікатор `L` слід застосовувати як у специфікаціях введення `scanf()`, так і в специфікаціях виведення функції `printf()`.

Введення символьних рядків за шаблоном. Функція `scanf()` підтримує ще один спосіб введення символьного рядка – зчитування послідовності символів, які належать (або, навпаки, не входять) до заданого набору.

Якщо замість літери-специфікатора у рядку формату після знака `%` записано групу символів у квадратних дужках `[...]` (її називають шаблоном), то з вхідного потоку будуть зчитані початкові символи, які збігаються зі заданими в шаблоні. При цьому великі та малі літери розрізняються, а пробільні символи розглядаються так само, як

усі інші символи. Ознакою кінця введення слугує перший символ, що не входить у заданий набір. Замість нього у введений рядок заноситься нуль-символ. Відповідний параметр у списку введення повинен задавати адресу ділянки оперативної пам'яті, куди будуть записані введені символи. Треба стежити, щоб виділена для рядка ділянка пам'яті мала достатній обсяг.

Ось приклад використання специфікації-шаблону.

```
char str[150];
scanf("%[AaBbCcDd]", str);
```

Нехай з клавіатури введено послідовність символів

```
Dabbca&Co
```

тоді в масив `str` буде занесено рядок `"Dabbca"` (з `'\0'` у кінці), а решта символів залишаться у буфері введення.

Підкреслимо, що символ пробілу та інші пробільні символи (наприклад, горизонтальну табуляцію) можна включати в шаблон і зчитувати в процесі введення, інакше перший роздільник (навіть на початку рядка) зупинить введення. Крім цього, більшість компіляторів підтримують (хоча це не є нормою стандарту) використання дефіса в записі еталонного набору. Наприклад, шаблон `[A-Za-z]` дає змогу ввести довільне слово, що буде складатись із великих чи малих латинських літер.

Якщо викликати `scanf()` із наступною специфікацією формату:

```
scanf("%[0-9A-Z ]", str);
```

то можна буде ввести з клавіатури довільний рядок символів, що містить цифри, великі латинські літери та символи пробілу, зокрема, такий:

```
32 PAGES    29875 CHARACTERS
```

Якщо ж першим символом шаблону записано знак `^`, то введення символьного рядка припиняється, коли зустрінеться перший символ, що збігається з одним із символів еталонного набору.

У разі використання в `scanf()` наступного набору символів-обмежувачів:

```
scanf("%[^.|:;/]", str);
```

зчитування даних буде припинено, коли у вхідному рядку з'явиться один із зазначених символів. Якщо введений рядок був таким:

```
456 | 67/56;   128.
```

то в `str` буде записано символьний рядок `" 456 "`, а всі інші символи, починаючи з `'|'`, залишаться не зчитаними.

Нестандартні варіанти введення. За допомогою звичайних символів, вказаних у рядку формату функції `scanf()`, можна задавати спеціальні форми запису вхідних даних і створювати зручний для користувача інтерфейс введення. Напрклад, функція:

```
scanf(" (%d,%d)", &x, &y);
```

забезпечує введення значень цілочислових координат (x, y), заданих парами чисел у круглих дужках. Вхідний рядок може бути таким:

```
(75,38)  (120,38)  (372,85)  (244,290)  (507,132)
```

Додаткові можливості для організації введення надає опція пропуску поля (символ * на початку специфікації). Проілюструємо це прикладом різних способів введення календарної дати. Нехай у програмі оголошено:

```
int day, month, year;
```

Якщо введення дати виконувати через функцію:

```
scanf("%d%d%d", &day, &month, &year);
```

то вхідні дані повинні відокремлюватись пробілами або іншими роздільниками, наприклад:

```
15 08 1990
```

Якщо ж додатково задати ширину кожного поля:

```
scanf("%2d%2d%4d", &day, &month, &year);
```

то дату можна буде записувати також єдиним рядком:

```
15081990
```

Вказавши розділову крапку в рядку формату, можна організувати введення дати у формі більш звичній для запису дат:

```
scanf("%d.%d.%d", &day, &month, &year);
```

Тепер дату треба вводити так:

```
15.08.1990
```

Кінець поля параметрів `day` і `mon` задає крапка, яка зчитується і пропускається.

У наступному прикладі використано рядок формату, який забезпечує пропуск одного символа після введення першого та другого числового значення:

```
scanf("%d%*c%d%*c%d", &day, &month, &year);
```

Це дає змогу вводити дату, в якій роль роздільника полів виконує довільний символ. Наприклад, дата може бути записана таким чином:

```
15/8/1990   або   15-08-1990
```

Запитання та завдання для самоконтролю

1. Що є першим параметром функцій `printf()` та `scanf()`? Яке його призначення?
2. Чим відрізняються списки даних функцій `printf()` та `scanf()`?
3. Які специфікації і в яких випадках застосовують для виведення цілих чисел? Чи можна їх використовувати для виведення значень дійсних типів?

4. Яку роль виконують елементи специфікацій [*ширина*] та [. *точність*] у разі виведення цілих чисел і в разі виведення дійсних чисел?
5. Які значення і в якій формі будуть виведені на екран після виконання наступних викликів `printf()`, якщо відповідні змінні оголошено так:

```
int  a=28,  b=-3,  ch='*';
double  w=0.505;
char  text[]="Вивчаємо функцію printf()";
```

1) `printf("%s:\n a=%u  b=%d  ch=%c\n", text, a, b, ch);`
2) `printf("\t3начення a => %d = %#o = %x\n", a, a, a);`
3) `printf("\t3начення b => %+d = %u = %#6X\n", b, b, b);`
4) `printf(" %f  |%10.2f  |%10.2e  |%10.4g  ", w, w*w, w+a,`
   `(double)a/(double)b);`

Перевірте себе, зреалізувавши програмно наведені виклики `printf()`.

6. Організуйте виведення на екран таких значень:
   1) значення натурального числа (відомо, що воно потрапляє в діапазон від 10 до 200) та в наступному рядку значень квадрата й куба цього числа;
   2) двох дійсних коренів квадратного рівняння, перед числовими значеннями яких має бути вказано: `Корені рівняння:`;
   3) поточної дати в формі: *День Найменування_місяця Рік*, а далі в дужках: *Найменування_дня_тижня*.
7. Як перевірити, чи функція `scanf()` успішно виконала введення вхідних даних?
8. В яких випадках виведення і введення даних необхідно застосовувати модифікатори `l` та `L`?
9. Оголосіть необхідні змінні та запишіть функцію `scanf()`, призначену для введення:
   1) тривалості часового інтервалу в секундах (довге беззнакове ціле);
   2) трьох дійсних коефіцієнтів квадратного рівняння;
   3) найменування товару, його ціни та кількості проданих виробів.
10. У програмі оголошено змінні:

```
unsigned int  num;
char  code[5];
double  res;
```

та записано функцію `scanf()` для введення значень цих змінних:

```
scanf("%u%s%lf", &num, code, &res);
```

У наведених нижче рядках вказано значення, які вводив користувач. Кожен рядок даних містить дві помилки – знайдіть їх.

1) `17,  AX-300  250`
2) `0X942  4561  0,124`
3) `10.8  X5-7  14.06g-7`

Розділ 6

ОПЕРАТОРИ

У цьому розділі:

- Класифікація операторів
- Оператори-вирази: оператор присвоєння, оператор виклику функції, порожній оператор
- Коротка та повна форма умовного оператора `if`
- Особливості оператора вибору `switch`
- Використання функції `exit()` для переривання програми
- Оператор циклу `for`: синтаксис і алгоритм роботи, використання операції "кома", вкладення циклів, внутрішнє оголошення параметрів
- Оператор циклу `while`
- Цикли з післяумовою – оператор `do-while`
- Група операторів переходу: оператор `goto`, оператор `break`, оператор `continue` та оператор `return`
- Генерування та використання псевдовипадкових чисел

Оператори – основні конструктивні компоненти програм, призначені для виконання встановлених дій. Саме через оператори реалізують послідовність кроків алгоритму розв'язання задачі. Мова С має невеликий, але потужний та ефективний за своїми функціональними можливостями набір операторів.

У перекладній літературі замість терміна *оператор* можна зустріти терміни *інструкція* або *команда*. Як зазначалось раніше, в книзі Б. Кернігана і Д. Рітчі [10] терміном *оператор* (*operator*) називаються *операції* над даними (тобто *знаки операцій*), а керуючі вказівки до дій називаються *інструкціями* (*instructions*).

За конструкцією всі оператори можна поділити на дві групи:

- *прості* оператори, до складу яких не входять інші оператори;
- *складені* оператори, обов'язковими компонентами яких є внутрішні (вкладені) оператори.

За характером дій, що виконуються операторами, виділяють такі групи:

- оператори-вирази;
- умовні оператори (`if`, `switch`);
- оператори циклу (`for`, `while`, `do-while`);
- оператори переходу (`goto`, `break`, `continue`, `return`).

Окремим видом складеного оператора є *блок*. Блоком у мові С називають групу довільних операторів, об'єднаних фігурними дужками: `{...}`. Після правої дужки блоку знак `;` не ставиться. Конструктивно блок розглядається як один оператор, він може бути записаний всюди, де синтаксис мови вимагає наявності оператора. Особливість блоків у тому, що всередині блоку перед першим оператором можна оголошувати внутрішні змінні. Такі змінні локалізуються в даному блоці, використовувати їх за межами блоку не можна (питання області видимості змінних детально розглядаються в розділі 12).

Наведемо два приклади блоків:

```
/* 1-й */
   { clrscr();  page++; }          /* оператори блоку */

/* 2-й */
   { double tmp;                   /* оголошення внутрішньої змінної */
     tmp=u;    u=v;    v=tmp;     /* обмін значеннями змінних u та v */
   }
```

Далі розглянемо властивості операторів кожної з класифікаційних груп.

6.1. Оператори-вирази

У мові С оператором вважається кожен допустимий вираз, що закінчується знаком крапка з комою `;`. Тобто всі наступні записи:

```
z=3.5*x;  /* 1 */        (a+b)*c;  /* 2 */        128;  /* 3 */
clrscr();  /* 4 */       g>MAX ? ++k1 : ++k2;  /* 5 */
```

синтаксично можна розглядати як *оператори-вирази*. Проте за означенням оператор повинен виконувати певну дію. Серед записаних операторів дії виконують: перший (присвоєння `z` значення виразу), четвертий (виклик функції очищення екрану) і п'ятий (збільшення значення змінної `k1` або `k2`). Другий і третій оператори дій не виконують, тому як оператори вони є беззмістовними (хоча в другому операторі обчислюється значення виразу `(a+b)*c`, але воно ніде не використовується і втрачається після завершення оператора).

Серед операторів-виразів виділяють дві групи:
- оператори присвоєння;
- оператори звертання до функцій.

Оператори присвоєння. Синтаксично оператори присвоєння є виразами, в яких останньою виконується операція присвоєння, а в кінці записано знак `;`. Можна використовувати кожну з форм операцій присвоєння: звичайне присвоєння, комбіноване присвоєння чи інкремент/декремент. Наведемо приклади таких операторів:

```
y=cos((x+2.5)/3-x*x);      /* звичайне присвоєння     */
k=m=n=arr[1];              /* послідовні присвоєння   */
num/=10;                   /* комбіноване присвоєння */
x++;     y--;              /* інкремент і декремент   */
```

Оператори виклику функцій. У мові С звертання до функції може використовуватись як операнд виразу (за умови, що функція повертає відповідне значення) або виступати окремим оператором.

Якщо функція не повертає ніякого значення (тип її значення `void`), то така функція є аналогом процедури в інших мовах програмування, і її можна викликати тільки як окремий оператор. Нижче наведено два приклади звертання до функцій, які не повертають значення:

```
randomize();               /* ініціалізує генератор випадкових чисел */
free(par);                 /* звільняє динамічну пам'ять */
```

Як окремий оператор може виступати і звертання до функції, яка повертає певне значення. У цьому випадку виконуються дії, що передбачені в функції, а значення, яке вона повертає, не застосовується і втрачається. Ось кілька прикладів:

```
scanf("%d%d", &g1, &g2);      /* повертає кількість введених даних */
puts("Кінець розрахунку.");   /* повертає ненульове значення */
strcpy(snew, buf);            /* повертає адресу скопійованого рядка */
```

Порожній оператор. Найпростішим у записі є *порожній* оператор – він позначається тільки знаком `;`. Порожній оператор не виконує ніяких дій, його застосовують у тих випадках, коли синтаксична конструкція вимагає запису оператора, але ніяких дій виконувати не треба. Найчастіше порожній оператор використовують в операторах циклу, рідше – в умовних операторах чи інших конструкціях. Приклади застосування порожнього оператора наведемо пізніше у відповідних задачах.

6.2. Умовні оператори

Умовні оператори реалізують розгалуження процесу виконання програми та дають змогу вибрати один з можливих варіантів продовження програми. Мова С має два види умовних операторів: `if` та `switch`.

6.2.1. Умовний оператор `if`

Короткий `if`. Оператор `if` має дві форми: скорочену та повну. Синтаксис скороченої форми умовного оператора такий:

```
if ( вираз )
    оператор
```

Вираз, записаний в дужках, визначає, чи буде виконуватись заданий оператор. Якщо значення виразу ненульове (умова істинна), то виконується внутрішній оператор `if`.

Якщо ж значення виразу дорівнює нулю (умова не справджується), то внутрішній оператор пропускається і керування відразу передається оператору, наступному за `if`.

У разі виконання оператора:

```
if (k==1)                    /* умова */
   sum=n1;                   /* оператор */
```

змінна `sum` набуде значення `n1` тільки тоді, коли `k` дорівнює `1`.

Внутрішнім оператором `if` може бути довільний оператор мови С: оператор-вираз, умовний оператор, оператор циклу чи оператор переходу. Якщо ж треба зробити залежною від заданої умови групу операторів, то їх оформляють як оператор-блок:

```
if (k==1) {                  /* умова */
   sum=n1;                   /* група операторів */
   n2=2*n1;
   flag=0;
}
```

Звернемо увагу на відступи, вирівнювання та розстановку дужок. Для читабельності програми внутрішні оператори `if` слід "втоплювати". Дужки блоку треба вирівнювати по вертикалі або розташовувати так, як це показано в попередньому прикладі.

Повний `if`. Синтаксис повної форми оператора `if` наступний:

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

Залежно від значення виразу-умови, записаного в дужках, даний оператор забезпечує виконання або *оператора_1*, або *оператора_2*, але ні за яких умов не будуть виконуватись обидва оператори. *Оператор_1* виконується, якщо значення виразу ненульове (істинне), а *оператор_2* – якщо значення виразу хибне (дорівнює нулю).

Розглянемо приклад:

```
if ( numb % 2 == 0 )
   puts ("Число парне");
else
   puts ("Число непарне");
```

Якщо умову оператора `if` змінити на протилежну, то внутрішні оператори слід переставити місцями. Попередній приклад можна записати і так:

```
if ( numb % 2 )                    /* тобто numb % 2 != 0 */
   puts ("Число непарне");
else
   puts ("Число парне");
```

Зауважимо також, що в деяких випадках замість оператора `if` можна застосовувати умовну операцію. Зокрема, для визначення парності числа можна використати ще й таку конструкцію:

```
printf("Число %s \n", numb%2 ? "непарне" : "парне");
```

Внутрішні *оператор_1* і *оператор_2* оператора `if` можуть бути довільними операторами мови, в тому числі порожніми операторами або ж блоками, які об'єднують групу операторів. Наприклад:

```
if (key!=0x1b)          /* натиснута клавіша - не Esc */
   Menu ();             /* викликати функцію меню */
else {
   fclose (infile);     /* закрити файл */
   return 0;            /* завершити поточну функцію */
}
```

Вкладення умовних операторів. Внутрішнім оператором `if` може бути вкладений `if`. Стандарт С-89 гарантував можливість 15 рівнів вкладення, а стандарт С-99 збільшив глибину вкладення до 127. Проте велика кількість вкладень заплутує програму і ускладнює пошук помилок, тому на практиці рідко застосовують глибину вкладень, більшу за 10. Оскільки оператори `if` мають дві форми: повну і неповну, то в разі вкладення операторів може виникнути ситуація, коли конструкцій `if` більше, ніж `else`. За правилами замовчування кожна `else`-частина умовного оператора пов'язується з найближчим `if`. Якщо ж `else` повинно відноситись до одного з попередніх `if`, то треба застосувати фігурні дужки `{}`, щоб виділити в блок попередню операторну частину. Наведемо два приклади.

```
/*  Приклад 1. Вкладення умовних операторів  */
if (c>b)                /* зовнішній if */
   if (beg==0)          /* вкладений if */
      max=c;
   else                 /* від вкладеного if */
      max+=c;
else                    /* від зовнішнього if */
   if (beg==0) {        /* вкладений неповний if */
      max=(c+b)/2;
      c=b;
   }
```

```
/*  Приклад 2. Вкладення умовних операторів  */
if (c<=b) {             /* зовнішній if */
    if (beg==0) {       /* вкладений неповний if */
       max=(c +b)/2;
       c=b;
    }
} else                  /* від зовнішнього if */
    if (beg==0)         /* вкладений if */
       max=c;
    else                /* від вкладеного if */
       max+=c;
```

Обидва фрагменти програми реалізують однакові дії. Зміна виразу умови в зовнішньому операторі `if` у прикладі 2 призвела до необхідності введення додаткових фігурних дужок, якими охоплюється вкладений неповний `if`. Без цих дужок конструкція першого `else` пов'язувалась би з умовою `beg == 0`, а не з умовою `c <= b`, як цього вимагає задача.

6.2.2. Оператор вибору `switch`

Цей оператор часто називають *перемикачем*. Він призначений для вибору одного з альтернативних варіантів у разі розгалуження процесу розв'язування задачі. Загальна структура оператора `switch` така:

```
switch ( вираз ) {
    case конст_1:  оператори
    case конст_2:  оператори
        . . .
    case конст_k:  оператори
    default:       оператори
}
```

Вираз, записаний в заголовку `switch`, може бути довільним виразом, що має цілочислове значення. Кожен варіант вибору починається службовим словом `case`, за яким вказується *константа вибору* (інша назва – *мітка*) даного варіанта: *конст_1*, *конст_2*, ... *конст_k*. Мітки можуть бути константами або константними виразами, що мають цілочислове чи символьне значення. Порядок запису варіантів вибору довільний, але значення всіх констант вибору повинні бути різними. Остання вітка оператора `switch`, яка починається службовим словом `default`, не є обов'язковою. У кінці оператора вибору записується права фігурна дужка `}`, після якої знак `;` не ставиться.

Оператор `switch` виконує наступні дії:

1) обчислюється значення виразу, який керує вибором варіанта продовження процесу виконання програми;
2) це значення послідовно порівнюється зі значенням констант вибору: *конст_1*, *конст_2*, ... *конст_k* – доки не буде знайдено відповідну константу;
3) якщо знайдено варіант, константна вибору якого збігається зі значенням виразу вибору, то виконуються *оператори* цього варіанта і всі наступні внутрішні оператори `switch`;
4) якщо значення *виразу* не збігається з жодною із констант вибору, а до складу `switch` входить альтернативний варіант `default`, то виконуються *оператори* цього варіанта;
5) якщо ж в операторі немає міток, які збігаються зі значенням виразу вибору та відсутній варіант `default`, то жодний із внутрішніх операторів не виконується, а керування передається наступному за `switch` оператору.

Продемонструємо особливості роботи оператора `switch` на прикладі наступної короткої програми.

```
/*************************************/
/*   Помилка в записі оператора switch  */
/*************************************/
#include <stdio.h>
int main(void)
{
    int key;
  printf("Натисніть одну з клавіш: A, B або C - ");
  key=getchar();                                  /* зчитування символа */
  switch (key) {
    case 'A':  printf ("\tКлавіша A. \n");
    case 'B':  printf ("\tКлавіша B. \n");
    case 'C':  printf ("\tКлавіша C. \n");
    default:   printf ("\tХибна клавіша! \n");
  }
  return 0;
}
```

Приклад виконання:

```
Натисніть одну з клавіш: A, B або C - B
     Клавіша B.
     Клавіша C.
     Хибна клавіша!
```

Наведений приклад ілюструє особливість оператора `switch`: після вибору варіанта розгалуження виконуються оператори даного варіанта й оператори всіх наступних варіантів. У більшості задач (зокрема в нашій програмі) виконання всіх операторів є помилковим, треба щоб виконувались тільки оператори вибраної вітки розгалуження. Перервати виконання операторів тіла `switch` можна оператором переходу `break`, який передає керування наступному за `switch` оператору. Отже, в попередньому прикладі оператор `switch` треба записувати так:

```
/*  Правильний запис оператора switch  */
switch (key) {
   case 'A':  printf ("\tКлавіша A. \n");  break;
   case 'B':  printf ("\tКлавіша B. \n");  break;
   case 'C':  printf ("\tКлавіша C. \n");  break;
   default:   printf ("\tХибна клавіша! \n");
}
```

Тепер результати виконання програми будуть правильними:

перший тест:

```
Натисніть одну з клавіш: A, B або C - B
     Клавіша B.
```

другий тест:

```
Натисніть одну з клавіш: A, B або C - M
    Хибна клавіша!
```

Зауважимо ще раз: за відсутності операторів переходу (`break`, `return`, `goto`) відбувається послідовне виконання всіх внутрішніх операторів `switch`, починаючи з вибраної вітки розгалуження. Тому кожна з альтернативних послідовностей операторів найчастіше завершується оператором `break`.

Розглянемо приклад фрагмента програми, що використовує оператор `switch` для виведення найменування дня тижня, заданого іменованою константою з переліку `wdays`.

```
/*  Виведення значення змінної перелікового типу  */
  enum wdays {Mon=1, Tue, Wed, Thu, Fri, Sat} day;
  int period;
. . .                  /* попередні дії */
day=period%6+1;
printf (" Цей день - ");
switch (day) {                                  /* визначення дня тижня */
  case Mon:  puts("понеділок");  break;
  case Tue:  puts("вівторок");   break;
    . . .                        /* аналогічно для інших днів тижня */
  case Sat:  puts("субота");     break;
}
```

Кожна вітка розгалуження в наведеному `switch` завершується оператором `break`. Хоча в останньому варіанті (`case Sat`) оператор `break` не є обов'язковим, його введено на випадок доповнення `switch` іншими вітками (наприклад, `case Sun` чи `default`).

Наступний приклад демонструє використання властивості наскрізного виконання внутрішніх операторів `switch`. У наведеному фрагменті програми залежно від значення змінної `file_number` закриваються один, два або три файли.

```
/*  Використання наскрізного виконання операторів switch  */
  int file_number;                  /* кількість відкритих файлів */
. . .                               /* попередні дії */
switch (file_number) {
  case 3:  fclose(ftmp);
  case 2:  fclose(fout);
  case 1:  fclose(fin);
}
```

У разі, якщо змінна `file_number` має значення `3`, то послідовно закриваються всі три файли: `ftmp`, `fout` та `fin`, якщо значення `file_number` дорівнює `2`, то закриваються два файли: `fout` та `fin`, інакше закривається тільки `fin`.

Внутрішніми операторами `switch` можуть бути довільні оператори, зокрема можна вкладати один оператор вибору в інший. У поданому нижче прикладі зовнішній оператор `switch` має три альтернативні вітки, вибором яких керує значення функції `menu()`. До складу операторів варіанта `case 3` входить вкладений оператор вибору, дія якого залежить від натисненої користувачем клавіші.

```
/*  Вкладення операторів switch  */

switch (menu()) {                        /* вибір через функцію menu() */
  case 1:  StartGame();   break;
  case 2:  PauseGame();   break;
  case 3:
    rept: printf ("Продовжити виконання?  Y(так)/N(ні) - ");
          switch (getchar()) {                  /* натиснено клавішу */
            case 'Y':  case 'y':  PlayGame();   break;
            case 'N':  case 'n':  exit(0);
            default:  goto rept;
          }
}
```

Звернемо увагу, що варіанти вкладеного оператора `switch` згруповані по два (їх може бути в групі й більше). Кожна група має спільну операторну частину. Перші два варіанти вкладеного `switch` закінчуються оператором `break`, який забезпечує перехід у зовнішній оператор вибору. Варіанти, позначені константами `'N'` та `'n'`, містять один внутрішній оператор – виклик функції `exit()`, яка завершує роботу програми.

6.2.3 Завершення роботи програми функцією `exit()`

Принагідно зупинимось на функції `exit()`, яку часто застосовують у практичних програмах. Призначення `exit()` – завершення роботи програми незалежно від того, чи дана функція викликається з `main()`, чи з іншої функції програми. При цьому автоматично закриваються всі відкриті в програмі файли, очищаються буфери потоків, звільняється виділена динамічна пам'ять тощо – як і при звичайному завершенні роботи програми. Для використання `exit()` до програми треба підключити стандартний заголовний файл `<stdlib.h>`.

Функція `exit()` має один параметр цілого типу, що задає значення, яке буде передане операційній системі після завершення роботи програми. Якщо функція викликається з параметром `EXIT_SUCCESS` (або `0`), то це вказує на нормальне закінчення роботи програми; всі інші значення цього параметра (зокрема `EXIT_FAILURE` або `1`) сигналізують про аварійне завершення програми. Макроконстанти `EXIT_SUCCESS` та `EXIT_FAILURE` задекларовані в заголовному файлі `<stdlib.h>`.

Здебільшого операційні системи ігнорують значення, яке повертає програма після завершення. Проте можна забезпечити аналіз цього значення та відповідну реакцію системи. Зокрема в командних bat-файлах MS DOS такий аналіз можна здійснювати через оператор if errorlevel.

6.3. Оператори циклу

Мова С підтримує три форми *операторів циклу*, які прийнято називати за їх ключовими словами: оператор `for`, оператор `while` та оператор `do-while`. Усі три оператори призначені для багаторазового (циклічного) виконання заданого набору дій.

6.3.1. Оператор `for`

Оператор `for` мови С – це потужний оператор з широкими функціональними можливостями, що дає змогу ефективно запрограмувати більшість видів циклічних процесів.

Синтаксис оператора `for` такий:

```
for ( вираз_1; вираз_2; вираз_3 )
    оператор
```

тут *вираз_1* – встановлює початкові значення змінних циклу, його називають виразом ініціалізації; *вираз_2* – задає умову виконання циклу, його називають виразом умови; *вираз_3* – виконує зміну значень змінних циклу, його називають виразом ітерації (приросту); *оператор* – довільний оператор мови, в т.ч. блок, який задає дії, що мають виконуватись кількаразово, його називають *тілом циклу*.

Оператор `for` працює наступним чином:

1) обчислюється *вираз_1*, який застосовують, щоб надати змінним циклу початкових значень;
2) обчислюється значення *виразу_2*, тобто перевіряється умова виконання циклу;
3) якщо значення *виразу_2* дорівнює нулю (умова хибна), то виконання циклу завершується і керування передається оператору, наступному за `for`;
4) якщо значення *виразу_2* ненульове (умова справджується), то виконується оператор тіла циклу (це може бути група операторів, об'єднана в блок);
5) обчислюється *вираз_3*, який застосовують для зміни значень параметрів циклу;
6) відбувається безумовний перехід до кроку 2.

Наведемо приклад використання оператора циклу `for` для виведення таблиці квадратів і кубів натуральних чисел від 35 до 55.

```
/*************************************************************/
/*  Таблиця квадратів і кубів натуральних чисел від 35 до 55  */
/*************************************************************/
#include <stdio.h>
int main(void)
{
    int n;
  printf ("   n      n*n     n*n*n \n");          /* шапка таблиці */
  for (n=35;  n<=55;  n++)
     printf ("%4d%8d%9ld \n", n, n*n, (long)n*n*n);
  return 0;
}
```

Результат виконання:

```
 n      n*n     n*n*n
35     1225     42875
36     1296     46656
37     1369     50653
      . . .
54     2916    157464
55     3025    166375
```

Параметром циклу в наведеному прикладі є змінна `n` – натуральне число, для якого обчислюються значення квадратів і кубів. Вираз ініціалізації (`n=35`) надає `n` початкового значення. Далі перевіряється вираз умови (`n<=55`), оскільки він справджується, то виконується оператор тіла циклу, який виводить на екран дані першого рядка таблиці. Після цього обчислюється вираз ітерації (`n++`) і змінна `n` набуває наступного значення. Знову перевіряється умова і процес повторюється. Коли після чергового збільшення значення `n` перевищить `55`, виконання циклу закінчується.

Для обчислення значень квадратів і кубів числа `n` у програмі виконується дво- і триразове множення цілих чисел. Такі операції пов'язані з небезпекою перевищення граничних значень цілого типу і вимагають посиленої уваги. Для заданого діапазону значень `n` (`35..55`) значення виразу `n*n` не перевищують верхню межу типу `int`, а значення `n*n*n` є більшими за `INT_MAX`, тому операнди цього виразу перед множенням перетворюються до типу `long`. У програмі використано вираз `(long)n*n*n`, в якому явне перетворення типу зазначене тільки для першого операнда. Наступні операнди будуть перетворюватись автоматично, оскільки асоціативність операції множення зліва направо.

Початкову частину оператора циклу `for(`*вираз_1*`;` *вираз_2*`;` *вираз_3*`)` називають *заголовком циклу*. Після заголовка знак `;` не ставлять (крім випадків порожнього оператора, про який мова йтиме далі). Всередині заголовка `;` застосовують для відокремлення між собою трьох складових виразів. Кожен із виразів: *вираз_1*, *вираз_2* і/або *вираз_3* може бути відсутній, але знак `;` вказується обов'язково. Наприклад, у попередній програмі можна було використати таку конструкцію:

```
   int  n=35;
for ( ;  n<=55;  n++)
   printf("%4d%8d%9ld \n", n, n*n, (long)n*n*n);
```

або організувати цикл наступним чином:

```
   int  n=35;
for ( ;  n<=55; ) {
   printf("%4d%8d%9ld \n", n, n*n, (long)n*n*n);
   n++;
}
```

У двох останніх прикладах ініціалізація `n` виконується перед циклом, тому в заголовку `for` ініціалізаційний вираз відсутній. У другому прикладі опущено також вираз ітерації, а нарощування `n` здійснюється в тілі циклу. За наочністю обидві версії

оператора for зі скороченим заголовком циклу поступаються першій, в якій використано цикл for з повним заголовком.

Останній приклад демонструє т. зв. безконечний цикл, в якому відсутня умова завершення. Щоб перервати роботу for, коли n досягне граничного значення, в тілі циклу введено оператор break.

```
    int n=35;
for ( ;  ; ) {                                          /* безконечний цикл */
    printf("%4d%8d%9ld \n", n, n*n, (long)n*n*n );
    if ( ++n > 55 )
        break;
}
```

Розглянемо ще одну програму, яка визначає суму заданої числової послідовності: $3+9+27+81+...+3^{15}$.

```
/*****************************************************/
/*  Обчислення суми ряду: 3+9+27+...+3^15. Варіант 1  */
/*****************************************************/
#include <stdio.h>
int main(void)
{
    int m;
    long num, sum;
  num = 3;
  sum = 0;
  for ( m=1;  m<=15;  m++ ) {
     sum = sum+num;
     num = 3*num;
  }
  printf ("Сума = %ld \n", sum);
  return 0;
}
```

Результат виконання:

```
Сума = 21523359
```

Цикл наведеної програми має три параметри: num – число-доданок, sum – сума елементів ряду і m – номер доданка. Ініціалізацію m виконано в заголовку for, а ініціалізацію num та sum – перед циклом. Нарощування m виконується в заголовку циклу, а значення змінної num збільшується в тілі циклу. Зручніше, коли вирази ініціалізації змінних циклу та їх ітераційних змін зібрані разом. У заголовку for це можна зробити за допомогою операції "кома".

Операція "кома". Операція послідовних обчислень, яка позначається знаком , (її назва – "кома"), має найнижчий пріоритет серед усіх операцій мови С. Вона призначена для об'єднання декількох послідовно записаних виразів (підвиразів) оператора for в

один загальний вираз. Операція "кома" гарантує, що підвирази будуть обчислюватися послідовно зліва направо.

З використанням операції "кома" попередню програму можна записати так:

```
/*****************************************************/
/*  Обчислення суми ряду: 3+9+27+...+3^15. Варіант 2  */
/*****************************************************/
#include <stdio.h>
int main(void)
{
    int  m;
    long  num, sum;
  for (num=3, sum=0, m=1; m <= 15; num *= 3, m++)
        sum += num;
  printf("Сума = %ld \n", sum);
  return 0;
}
```

У заголовку `for` операція "кома" об'єднує три підвирази виразу ініціалізації: `(num=3, sum=0, m=1)` ⇒ *вираз_1*, а у виразі ітерації "комою" пов'язано два підвирази: `(num *= 3, m++)` ⇒ *вираз_3*.

Якщо до виразу ітерації долучити ще й вираз нарощування суми, то оператор `for` можна записати, наприклад, так:

```
/*  Фрагмент обчислення суми ряду: 3+9+27+...+3^15. Варіант 3  */
for (m=1, num=sum=3;  m<15;  num*=3, sum+=num, m++)
            ;
```

Синтаксично цей оператор цілком правильний. Результат його виконання дасть значення суми `sum=21523359`, що повністю збігається з результатами роботи двох попередніх варіантів програми. Проте за наочністю останній оператор `for` є найгіршим, бо не виділяє основного призначення циклу – обчислення суми. Оскільки операцію збільшення суми `sum+=num` введено у заголовок `for`, то операторів у тілі циклу не залишилось, а саме тіло позначається порожнім оператором `;` (за синтаксисом `for` завжди повинен мати оператор тіла циклу). Знак `;` доцільно записувати в окремому рядку програми (як у наведеному прикладі), щоб візуально підкреслити, що внутрішній оператор циклу порожній.

Звернемо увагу на порядок запису трьох підвиразів виразу ітерації. Операція "кома" забезпечує їх послідовне виконання: спочатку `num*=3`, потім `sum+=num` і останнім виконується `m++`. Цей порядок відрізняється від порядку дій, що застосовувався у двох попередніх варіантах програми. Тому в третьому варіанті `for` змінено вирази ініціалізації та умови виконання циклу.

Найчастіше помилковими бувають саме граничні умови циклів. Потрібно завжди уважно перевіряти, чи початкові значення, умова та ітераційні прирости строго охоплюють увесь заданий діапазон значень параметрів циклу.

Вкладення циклів. У багатьох програмах у тілі циклу доводиться організовувати внутрішні циклічні дії. Такі цикли називають *вкладеними*. Наступна програма використовує вкладення циклів для виведення на екран пірамідки літер.

```
/*********************************/
/*  Побудова літерної піраміди   */
/*********************************/

#include <stdio.h>
#define RMAX 8                  /* висота піраміди */
int main(void)
{
    int row, left, right, pos;
  for (row=1, left=right=RMAX;  row<=RMAX;  left--, right++, row++) {
      for (pos=1;  pos<left;  pos++)
          putchar (' ');                /* виведення символа пробілу  */
      for ( ;  pos<=right;  pos++)
          putchar ('A'+row-1);          /* виведення літер рядка row  */
      putchar ('\n');                   /* перехід до нового рядка    */
  }
  return 0;
}
```

Результат виконання:

```
       A
      BBB
     CCCCC
    DDDDDDD
   EEEEEEEEE
  FFFFFFFFFFF
 GGGGGGGGGGGGG
HHHHHHHHHHHHHHH
```

У наведеній програмі керуючим параметром зовнішнього циклу є змінна `row`, значення якої відповідає номеру рядка виведення. Змінні `left` та `right` задають номери позицій початку і кінця літерного рядка. Початково вони збігаються, бо в першому рядку виводиться тільки один символ. Після виведення кожного рядка `left` зменшується, а `right` зростає на 1. Тіло зовнішнього оператора `for` складається з двох вкладених операторів `for` та оператора виклику функції виведення символа нового рядка `putchar('\n')`. Внутрішні оператори `for` формують на екрані рядок літер: перший відтворює початкові пробіли, а другий виводить (починаючи від позиції `left` до позиції `right`) літеру, яка відповідає номеру рядка. Вираз `'A'+(row-1)` задає літеру, код якої на `row-1` більший за код літери `'A'`.

Попередню програму можна дещо скоротити, якщо для виведення рядка замість двох внутрішніх операторів `for` використати один і застосувати умовний вираз для вибору символа, який має бути відображений: пробіл чи задана літера. Реалізувати

таке виведення можна як через функцію `putchar()`, так і через функцію `printf()`:

```
for (pos=1; pos<=right; pos++)
   printf("%c", pos<left ? ' ' : 'A'+(row-1));      /* вибір символа */
```

У разі вкладення операторів циклу слід обов'язково дотримуватись правил відступів і вирівнювань, без яких читабельність програми різко погіршується. Більшість компіляторів С практично не обмежують глибину вкладення, але аналізувати і налагоджувати цикли з десятками вкладень надто складно. Краще виділити окремі частини циклічного процесу в функції чи застосувати інші алгоритми та способи програмування.

Внутрішнє оголошення параметрів `for`. Стандарт С-99 надав додаткову можливість для операторів `for` – один або декілька параметрів `for` можна оголошувати у виразі ініціалізації заголовка циклу. Такі змінні будуть локальними змінними циклу, тобто їх можна використовувати тільки в межах даного `for`.

Внутрішні оголошення змінних циклу дозволені в С++. Програмісти часто застосовують їх, коли для організації циклу треба ввести тимчасову змінну. Тепер це можна робити і в системах програмування, що реалізують С-99.

Записана далі програма виводить на екран набір символів псевдографіки з другої половини ASCII-таблиці. Ці символи застосовують для формування рамок, таблиць, меню та інших нескладних графічних побудов у текстових режимах роботи відеосистеми. Параметр `ch` у циклі `for` оголошено як внутрішній параметр циклу.

```
/*************************************************/
/*  Виведення символів псевдографіки ASCII-таблиці  */
/*************************************************/

#include <stdio.h>
int main(void)
{
  printf("\t\t\t Символи псевдографіки: \n");
  for (int ch=176; ch<=223; ch++)        /* внутрішнє оголошення ch */
     printf("%c (%3d)   ", ch, ch);
  return 0;
}
```

Результат виконання:

```
                    Символи псевдографіки:
▒ (176)  ▓ (177)  █ (178)  │ (179)  ┤ (180)  ╡ (181)  ╢ (182)  ╖ (183)
╕ (184)  ╣ (185)  ║ (186)  ╗ (187)  ╝ (188)  ╜ (189)  ╛ (190)  ┐ (191)
└ (192)  ┴ (193)  ┬ (194)  ├ (195)  ─ (196)  ┼ (197)  ╞ (198)  ╟ (199)
╚ (200)  ╔ (201)  ╩ (202)  ╦ (203)  ╠ (204)  ═ (205)  ╬ (206)  ╧ (207)
╨ (208)  ╤ (209)  ╥ (210)  ╙ (211)  ╘ (212)  ╒ (213)  ╓ (214)  ╫ (215)
╪ (216)  ┘ (217)  ┌ (218)  ▒ (219)  ▄ (220)  ▌ (221)  ▐ (222)  ▀ (223)
```

Реально в програмах, відкомпільованих у середовищі Borland C, областю дії змінної, оголошеної всередині оператора `for`, є блок, в якому розташований цей оператор (блоком може бути й усе тіло функції). Тому таку змінну можна використовувати за межами `for` до кінця поточного блоку.

6.3.2. Оператор while

Іншим видом оператора циклу мови С є оператор `while`, що має такий синтаксис:

```
while (вираз)
    оператор
```

тут *вираз* – довільний вираз, що задає умову виконання циклу; *оператор* – довільний оператор мови (зокрема блок), що формує тіло циклу.

Оператором `while` виконуються наступні дії:

1) обчислюється значення *виразу*, тобто перевіряється умова виконання циклу;
2) якщо значення *виразу* дорівнює нулю (умова хибна), то виконання циклу завершується і керування передається оператору, наступному за `while`;
3) якщо значення *виразу* ненульове (умова істинна), то виконується оператор тіла циклу;
4) відбувається повернення до п. 1 для наступної перевірки умови виконання циклу.

Наведемо приклад фрагмента програми, в якому оператор `while` використовується для пропуску початкових пробільних символів у рядку, який вводиться з клавіатури.

```
/*  Пропуск пробільних символів  */
    int symb = ' ';
  while (symb == ' ')
     symb = getchar();                    /* зчитування символа */
  printf ("Перший символ слова - %c\n", symb);
```

Перед початком циклу змінній `symb` присвоєно значення символа пробілу `' '`, тому перша перевірка умови `while` справджується і відбувається введення символа з клавіатури. Зчитаний символ перевіряється, і якщо це символ пробілу, то вводиться наступний символ. Цикл завершиться, коли буде зчитано перший непробільний символ.

Наступний приклад демонструє використання оператора `while` у програмі, яка реалізує таку задачу: з клавіатури послідовно вводяться значення координат (x, y) групи точок – треба визначити точку, найбільш віддалену від початку координат. Для організації циклу в умовному виразі оператора `while` аналізується значення, яке повертає функція `scanf()`, тобто кількість успішно введених даних. Коли замість числа в рядку введення зустрінеться літера `Z` (або інший нецифровий символ), `scanf()` фіксує помилку, повертає `0`, і виконання циклу завершується.

```
/*****************************************/
/*  Визначення найбільш віддаленої точки  */
/*****************************************/

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

int main (void)
{
     double x, y, d, xmax, ymax, dmax;
     int n, nmax;
```

```
    n=1;
    dmax=0.0;                              /* змінна максимальної віддалі */
    printf("Координати точок (завершення - літера Z):\n%d.  ", n);
    while (scanf("%lf %lf", &x, &y)==2) {
       d=sqrt(x*x+y*y);                        /* віддаль поточної точки */
       if (d>dmax) {
          dmax=d;   nmax=n;                          /* номер точки */
          xmax=x;   ymax=y;                          /* та її координати */
       }
       printf("%d.  ", ++n);
    }
    printf("Найдальша точка - %d (%1.1f, %1.1f)\n", nmax, xmax, ymax);
    return 0;
}
```

Приклад виконання програми:

```
Координати точок (завершення - літера Z):
1.  28.92  14.65
2.  33.6   10.09
3.  28.49  25.33
4.  19.46  22.8
5.  31.82  7.345
6.  Z

Найдальша точка - 3 (28.5, 25.3)
```

6.3.3. Оператор do-while

Оператор `do-while` називають *оператором циклу з постумовою*. Синтаксис цього оператора такий:

```
do
   оператор
while (вираз);
```

Як і в операторі циклу `while`, в `do-while` *оператор* може бути довільним оператором, в т. ч. блоком, що формує тіло циклу, а *вираз* – довільний вираз, що задає умову виконання циклу.

На відміну від операторів `for` та `while`, в яких вираз-умова перевіряється відразу на початку циклу (тому оператор тіла циклу може не виконуватись жодного разу), в операторі `do-while` спочатку виконується оператор тіла циклу, а вже потім здійснюється перевірка умови продовження циклу. Використання оператора `do-while` зручне у випадках, коли значення виразу-умови залежить від результатів виконання тіла циклу. Приміром, наведений у попередньому параграфі фрагмент програми, що виконує пропуск початкових пробільних символів у рядку введення, раціональніше запрограмувати через оператор `do-while`:

```
/*  Пропуск пробільних символів - використання do-while  */
  int symb;
do {
   symb=getchar();                          /* зчитування символа */
} while (symb==' ');
printf ("Перший символ слова - %c\n", symb);
```

Попередній приклад цього програмного фрагмента з використанням оператора `while` вимагав присвоєння змінній `symb` початкового значення. Цикл `do-while` цього не потребує, оскільки спочатку зчитується символ, а вже потім проводиться його перевірка. Тіло єдиного вкладеного оператора охоплено фігурними дужками тільки для наочного виділення циклу `do-while`.

У наступному прикладі оператор `do-while` використано для генерування парного випадкового числа (бібліотечні функції генерування рівномірно розподілених випадкових чисел описані далі в параграфі 6.5).

```
/*  Генерування парного випадкового числа  */
  int rn;
do {
   rn=rand();                          /* звертання до функції-генератора */
} while (rn%2);                        /* тобто  rn%2!=0  - число непарне */
```

Цикл завершиться, коли значення виразу `rn%2` дорівнюватиме `0`, тобто `rn` буде парним числом.

Останній приклад – фрагмент програми, в якому реалізовано перевірку, чи введене з клавіатури значення потрапляє в заданий діапазон.

```
/*  Контроль правильності введення. Варіант 1  */
  int v;
printf("Вкажіть швидкість (%d - %d): ", VMIN, VMAX);
do {
   scanf("%d", &v);
   if (v<VMIN || v>VMAX) {
      printf ("Неправильне значення. Повторіть введення: ");
      v=0;
   }
} while (v==0);            /* доки не буде введене правильне значення */
```

6.4. Оператори переходу

У мові С реалізовано чотири *оператори переходу* (їх ще називають *керуючими* операторами): `goto`, `break`, `continue` та `return`. Усі вони змінюють послідовний порядок виконання операторів і передають керування не наступному, а певному встановленому оператору програми.

6.4.1. Оператор goto

Оператор `goto` передає керування іншому оператору програми, відзначеному заданою міткою. Синтаксис `goto` такий:

```
goto мітка;
   . . .                    /* оператори програми */
мітка : оператор
```

Міткою у мові С може бути довільний ідентифікатор, який не збігається з іншими ідентифікаторами програми. Мітка відокремлюється від оператора, який вона позначає, двокрапкою : (після : можна записувати символи пробілу). Загалом у С не вимагається, щоб мітка стояла перед оператором, вона може бути записана в довільному місці програми, наприклад, перед дужкою } кінця блоку або функції.

Використаємо `goto` для перевірки правильності введеного значення:

```
/*  Контроль правильності введення. Варіант 2  */
  int v;
printf ("Вкажіть швидкість (%d - %d): ", VMIN, VMAX);
inv: scanf("%d", &v );
if (v<VMIN || v>VMAX) {
   printf ("Неправильне значення. Повторіть введення: ");
   goto inv;                    /* перехід для повторного введення */
}
```

У С-програмах рідко використовують оператор `goto`. Вважається, що він заплутує програму і робить її малозрозумілою. Відшукати у великій програмі місце, позначене міткою, нелегко – тим більше, що перехід можна виконувати як у напрямку кінця програми, так і в напрямку її початку. Крім того, мова С практично не накладає обмежень на переходи: можна перестрибувати з блоку в блок, заходити всередину тіла циклу тощо. Єдина заборона – не можна переходити з однієї функції в іншу. Тому в разі складної організації переходів можуть виникнути непередбачувані ситуації в процесі виконання програми. Найкраще відмовитись від `goto`, замінивши його іншими операторами переходу (`break`, `continue` чи `return`), застосувати оператори циклу для повторного виконання заданих дій або використати інші програмні конструкції.

6.4.2. Оператор break

Оператор переходу `break` призначений для переривання роботи оператора вибору `switch` і всіх трьох операторів циклу: `for`, `while` та `do-while`.

Особливості застосування `break` в операторі вибору `switch` обговорювались раніше, коли розглядалась робота цього оператора. Нагадаємо тільки, що `break` перериває послідовне виконання внутрішніх операторів `switch` і передає керування оператору, наступному за оператором вибору.

У разі, якщо оператор `break` викликається в тілі оператора циклу, то виконання оператора циклу відразу припиняється (без перевірки умов завершення) і керування переходить до оператора, наступного за оператором циклу.

Застосовуючи оператор `break`, попередній приклад контролю коректності введеного значення можна записати так:

```
/*  Контроль правильності введення. Варіант 3  */
  int v;
printf ("Вкажіть швидкість (%d - %d): ", VMIN, VMAX);
while (1) {                              /* безконечний цикл */
   scanf("%d", &v);
   if (v>=VMIN && v<=VMAX)               /* якщо значення правильне, то */
      break;                             /* завершуємо цикл введення */
   printf ("Неправильне значення. Повторіть введення: ");
}
```

У наведеному програмному фрагменті використано безконечний цикл:

```
while (1) {
   . . .        /* тіло циклу */
}
```

Переривання роботи циклу здійснює оператор `break`, який виконується, коли введене значення потрапляє в допустимий діапазон [VMIN, VMAX].

6.4.3. Оператор continue

Оператор переходу `continue` може використовуватись тільки в тілі операторів циклу. Його призначення – перервати поточне виконання операторів тіла циклу і перейти до нової ітерації циклу. При цьому всі оператори тіла циклу, записані після `continue`, пропускаються. Якщо `continue` виконується в операторі `for`, то відразу після нього обчислюється вираз ітерації, а потім перевіряється умова виконання циклу. Якщо ж `continue` виконується всередині циклів `while` або `do-while`, то відразу відбувається перехід на наступну перевірку умови виконання циклу.

Проілюструємо використання оператора `continue` прикладом фрагмента програми, в якому визначається середнє значення додатних елементів масиву дійсних чисел.

```
/*  Обчислення середнього значення додатних елементів масиву  */
    double arr[100], sum;
    int i, n;
  . . .                                          /* попередні дії */
  for (sum=0.0, i=n=0;  i<100;  i++) {
     if (arr[i] < 0)  continue;       /* від'ємні елементи пропускаємо */
     sum+=arr[i];
     n++;
  }
  printf ("Середнє значення - %1.3f \n", n > 0 ? sum/n : 0 );
```

У наведеному прикладі оператор `continue`, по-перше, підкреслює, що від'ємні елементи масиву не аналізуються, по-друге, дає змогу уникнути внутрішнього блоку `{...}` в операторі `if`, який мав би об'єднувати два оператори нагромадження сум.

6.4.4. Оператор return

Оператор `return` завершує роботу функції, в якій він виконується. Керування програмою повертається до оператора, з якого була викликана ця функція.

Оператор `return` має дві форми:

```
return;                    /* без повернення значення */
return вираз;              /* зі значенням, що буде повернене */
```

Перша форма застосовується в функціях, які не повертають значення – такі функції повинні мати тип `void`. Друга форма оператора `return` застосовується, коли функція повинна повернути значення певного типу. В цьому випадку *вираз* задає значення, яке поверне функція. Оператор `return` обчислює значення вказаного виразу і перетворює його до типу, заданого в оголошенні функції, згідно з правилами перетворення типів в операціях присвоєння. Отримане значення передається в точку виклику функції.

Функція `main()` розпочинає і закінчує роботу всієї програми. Тому виконання оператора `return` всередині `main()` викликає завершення роботи програми. Наведемо приклад використання оператора `return` у задачі визначення дійсних коренів квадратного рівняння. У разі від'ємного значення дискримінанта `d` програма виводить повідомлення про відсутність дійсних коренів і завершує роботу. Інакше обчислюються два корені `x1` і `x2` та друкуються їх значення.

```
/*******************************************/
/*  Обчислення коренів квадратного рівняння  */
/*******************************************/

#include <stdio.h>
#include <math.h>
int main(void)
{
     double  a, b, c, d, x1, x2;
   printf("\nКоефіцієнти a, b, c:  ");
   scanf("%lf%lf%lf", &a, &b, &c);
   d=b*b-4*a*c;                          /* дискримінант */
   if (d<0) {
      printf("\tДійсних коренів немає. \n");
      return 0;
   }
   x1=(-b+sqrt(d))/(2*a);
   x2=(-b-sqrt(d))/(2*a);
   printf ("\tКорені: x1 = %1.4g  x2 = %1.4g\n", x1, x2);
   return 0;
}
```

Приклади виконання:

перший тест:

```
Коефіцієнти a, b, c:  3.44  5.86  7.21
    Дійсних коренів немає.
```

другий тест:

```
Коефіцієнти a, b, c:  4.28  6.05  -10.84
    Корені: x1 = 1.035  x2 = -2.448
```

третій тест:

```
Коефіцієнти a, b, c:  3  -6  3
    Корені: x1 = 1  x2 = 1
```

6.5. Використання псевдовипадкових чисел

У програмах ігрових задач, імітаційного моделювання та багатьох інших широко використовують т. зв. *псевдовипадкові числа*. Стандарт мови С підтримує дві бібліотечні функції: `rand()` та `srand()`, призначені для *генерування* послідовностей рівномірно розподілених випадкових чисел. У бібліотеку системи програмування Borland C входять ще дві функції: `random()` та `randomize()`, які доповнюють можливості стандартних функцій і спрощують програмування процесів, пов'язаних із використанням псевдовипадкових чисел. Всі функції роботи з псевдовипадковими числами оголошені в стандартному заголовному файлі `<stdlib.h>`.

Термін *псевдовипадкові* відображає той факт, що згенеровані числа реально не є повністю випадковими, а формуються програмно за певними алгоритмами [13]. Рівномірність розподілу псевдовипадкових чисел означає, що в разі генерування великого набору цих чисел, ймовірність появи кожного з можливих значень приблизно однакова, тобто згенеровані числа рівномірно заповнюють інтервал допустимих значень.

Функція `rand()` не має параметрів і повертає значення з типом `int`:

```
int  rand(void);
```

Значенням функції є псевдовипадкове число з проміжку `0..RAND_MAX`, де макроконстанта `RAND_MAX` дорівнює значенню `INT_MAX`. Кожне звертання до функції `rand()` у процесі виконання програми викликає генерування нового псевдовипадкового числа з рівномірним законом розподілу. Роботу функції ілюструє подана нижче програма, в якій генеруються і виводяться на екран `N` послідовних псевдовипадкових чисел.

```
/*****************************************************/
/*  Генерування послідовності псевдовипадкових чисел  */
/*****************************************************/
#include <stdio.h>
#include <stdlib.h>
#define  N 8                          /* кількість чисел */
```

```
int  main(void)
{
    int  n, ran;
  for (n=0;  n<N;  n++) {       /* цикл генерування і виведення чисел */
     ran=rand();
     printf("%d   ", ran);
  }
  return 0;
}
```

Приклад виконання програми:

```
346   130   10982   1090   11656   7117   17595   6415
```

У разі повторного запуску даної програми на виконання виведена на екран послідовність буде складатись із тих самих чисел. Причина в тому, що функція `rand()` починає генерування чисел із єдиного встановленого значення. Щоб змінити стартове значення функції `rand()` (його ще називають *затравкою*), використовують функцію `srand()`, яка має єдиний параметр – число, що слугуватиме затравкою для наступного виклику `rand()`:

```
void  srand(unsigned seed);
```

Якщо в попередній програмі перед оператором `for` записати виклик функції:

```
srand(число);
```

і перед кожним запуском програми змінювати це число, то кожного разу буде генеруватись інша послідовність псевдовипадкових чисел.

Найчастіше затравку для `rand()` формують з даних системного таймера, що забезпечує її автоматичне оновлення для кожної реалізації програми. У наступній версії програми генератор випадкових чисел запускається числом, яке формується з двох однакових за кодом байтів, що дорівнюють молодшому байту змінної `curt`. У `curt` попередньо заноситься значення поточного системного часу. Для цього використано бібліотечну функцію `time()`, єдиним параметром якої є адреса змінної з типом `long`. (Функції дати та часу оголошені в заголовному файлі `<time.h>`, їх характеристики наведено в табл. Д2.6 Додатка 2).

```
/*******************************************************/
/*  Генерування різних наборів псевдовипадкових чисел  */
/*******************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define  N 10                          /* кількість чисел */
int  main(void)
{
     int  n;
     long  curt;
```

```
    unsigned  seed;
   time (&curt);                              /* визначення системного часу */
   seed = (unsigned)(curt << 8 | curt & 0xff);     /* формування затравки */
   srand (seed);                        /* запуск генератора зі значення seed */
   for (n=0;  n<N;  n++)                    /* цикл друку згенерованих чисел */
      printf( "%d   ", rand() % 500 );
   return 0;
}
```

Кожна реалізація даної програми генерує іншу послідовність з `N` псевдовипадкових рівномірно розподілених чисел. Діапазон значень чисел звужено до `0..500-1` через використання виразу `rand()%500`.

Перший тестовий запуск програми дав такий результат:

```
134   30   343   357   366   62   177   49   252   487
```

Повторний запуск згенерував послідовність:

```
403   339   89   9   82   116   181   397   41   280
```

А після третього запуску отримано числа:

```
296   62   392   186   418   73   389   427   306   246
```

Як вже зазначалось, у бібліотеці Borland C є ще дві додаткові функції для звертання до генератора псевдовипадкових чисел: `random()` та `randomize()`. Насправді, вони оформлені як макроси, що звертаються до функцій `rand()` та `srand()`.

Перша з цих функцій:

```
int  random ( int hval );
```

повертає псевдовипадкове число з діапазону `0..hval-1`.

Друга функція:

```
void  randomize ( void );
```

не має параметрів і не повертає значення. Вона призначена для затравки функції генератора псевдовипадкових чисел поточним значенням системного таймера, подібно до того, як це зроблено в попередній програмі (але без зсування байтів).

Подана нижче ігрова програма "*Вгадай число*" використовує названі функції для формування цілого випадкового числа з проміжку `RMIN..RMAX`, яке гравець повинен вгадати за `K` спроб, користуючись висвітленими підказками.

```
/*****************************/
/*     Гра  "Вгадай число"     */
/*****************************/

#include <stdio.h>
#include <stdlib.h>
#define  RMIN 1                              /* найменше число  */
#define  RMAX 100                            /* найбільше число */
#define  K 6                                 /* кількість спроб */
```

```
int main (void)
{
   int numb, answ, k;
  randomize();                          /* затравка генератора від таймера */
  numb = RMIN + random(RMAX-RMIN+1);                  /* випадкове число */
  printf("\n\t*****   Задумано число з проміжку %d..%d - \n"
    "\t\t  вгадайте його за %d спроб \t*****\n", RMIN, RMAX, K);
  for (k=1;  k<=K;  k++) {              /* цикл вгадування за К спроб */
      printf("Спроба %d - ", k);
      scanf("%d", &answ);
      if (answ==numb) {                                /* число вгадано */
         printf("\n\t\t ***   %d - Ви вгадали!!!   ***", numb);
         return 0;
      }
      if (answ>numb)                              /* виведення підказки */
        printf("\t\t - завелике\n");
      else
        printf("\t\t - замале\n");
  }
  printf("\n\t\t***   Ви не вгадали, це число - %d   ***", numb);
  return 0;
}
```

Приклад виконання програми:

```
     *****   Задумано число з проміжку 1..100 -
             вгадайте його за 6 спроб        ****
Спроба 1 - 33
           - замале
Спроба 2 - 78
           - завелике
Спроба 3 - 64
           - замале
Спроба 4 - 75

          ***   75 - Ви вгадали!!!   ***
```

Запитання та завдання для самоконтролю

1. На які групи поділяють оператори мови С? Яким є призначення операторів кожної класифікаційної групи?
2. В яких випадках слід застосовувати умовний оператор `if`? Коли доцільніше використати оператор вибору `switch`? Наведіть приклади.
3. З чого формується заголовок циклу `for`? Що складає тіло циклу?
4. У чому основна відмінність операторів циклу `while` та `do-while`?
5. Яке призначення операторів `break` та `continue`?

6. Як можна перервати роботу програми? Що виконує оператор `return`?
7. Яке значення отримає змінна `fst` після виконання вказаних умовних операторів?

```
if (a>b)
   if (a>c)
      fst=a;
   else
      fst=c;
else  if (b>c)
         fst=b;
      else
         fst=c;
```

8. Проаналізуйте наведену програму. Які дії вона виконує?

```
/*  Калькулятор  */
#include <stdio.h>
#include <stdlib.h>
#define END '0'
#define ERRS -1
void main()
{
    int a=50, b=6;
    int znak=0, c;
  while (1) {
    printf ("\n\nЗнак операції: +, -, *, /, %  "
            "(0 - для завершення) => ");
    scanf(" %c", &znak);
    switch (znak) {
      case '+': c=a+b;  break;
      case '-': c=a-b;  break;
      case '*': c=a*b;  break;
      case '/': c=a/b;  break;
      case '%': c=a%b;  break;
      case END: exit(1);
      default: c=ERRS;
    }
    if (c!=ERRS)
      printf("\n%d%c%d=%d\n", a, znak, b, c);
    else
      printf ("Невірний знак (%c)\n", znak);
  }
}
```

Перевірте правильність своїх міркувань, протестувавши виконавчий код програми. Доповніть текст програми, щоб значення операндів `a` і `b` теж вводились з клавіатури.

9. Напишіть три версії фрагмента програми, який виконує виведення на екран усіх парних чисел з проміжку `[c, d]`, де `c` і `d` – два довільних цілих числа. Використайте в кожній з цих версій іншу форму оператора циклу.

10. Яким буде значення змінної `smn` після виконання наступного фрагмента програми?

```
unsigned  nmb = 45065,  smn = 0;
do {
      smn += nmb % 10;
      nmb /= 10;
} while(nmb > 0);
```

Запрограмуйте подані нижче задачі, використовуючи різні форми умовних операторів та операторів циклу

11. З клавіатури ввести значення трьох дійсних змінних: `z1`, `z2` та `z3`. Поміняти місцями значення цих змінних так, щоб значення `z1` було найменшим, а `z3` – найбільшим. Вивести на екран нові значення `z1`, `z2` та `z3`.

12. Ввести значення довгого натурального числа. Перевірити, чи це число є простим (простим вважається число, яке не ділиться цілочислово на інші натуральні числа, крім 1). Вивести відповідне повідомлення.

13. Задано два цілих беззнакових числа. Визначити найменше спільне кратне цих чисел.

14. З клавіатури ввести значення дійсного числа x. Обчислити значення квадратного кореня з цього числа: $y = \sqrt{x}$ з точністю $\varepsilon = 10^{-5}$, використовуючи рекурентне співвідношення:

$$y_0 = 1; \quad y_i = \frac{1}{2}\left(y_{i-1} + \frac{x}{y_{i-1}}\right), \quad i = 1, 2, 3, \ldots$$

На екран вивести значення x і y, а також кількість ітерацій циклу обчислення кореня. Порівняти отриманий результат із значенням, яке повертає стандартна бібліотечна функція `sqrt()`.

15. Визначити точку (x, y), в якій функція двох змінних $z = \cos 3x/(\sin^2 y + 1)$ набуває мінімального значення. Для цього обчислити значення функції в усіх точках координатної сітки $x \times y$ розміром $n \times n$, у межах якої x змінюється від 0 до $\pi/2$, а y – від $\pi/8$ до 2, значення n ввести з клавіатури.

16. Згенерувати і вивести на екран 15 випадкових трицифрових чисел, кожне з яких повинно містити хоча б одну цифру 7.

17. Побудувати на екрані квадрат, який буде складатись із n^2 символів 0 та 1, записаних так, щоб одиниці ділили цей квадрат на 4 однакових частини (див. приклади).

```
0 0 1 0 0
0 0 1 0 0
1 1 1 1 1      (n = 5)
0 0 1 0 0
0 0 1 0 0
```

```
0 0 1 1 0 0
0 0 1 1 0 0
1 1 1 1 1 1      (n = 6)
1 1 1 1 1 1
0 0 1 1 0 0
0 0 1 1 0 0
```

18. Обчислити і вивести на екран 7 початкових цифр дробової частини значення $\pi/4$, використовуючи ряд Лейбніца:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \ldots$$

Вказати, скільки доданків ряду враховано в сумі. *Увага*: для досягнення заданої точності необхідно просумувати більше, ніж `INT_MAX` елементів ряду.

Розділ 7

ВКАЗІВНИКИ

У цьому розділі:

- Поняття вказівника, оголошення вказівників
- Визначення адреси змінної (операція &) та звертання до даних через вказівники (операція *)
- Особливості застосування вказівників з кваліфікатором `const`
- Операції над вказівниками (адресна арифметика): присвоєння, порівняння, збільшення/зменшення значення вказівника, віднімання вказівників
- Характерні вирази з вказівниками
- Нетипізовані вказівники, перетворення базових типів вказівників

Широке використання вказівників у С-програмах – одна з визначальних рис мови, що особливо приваблює програмістів-професіоналів. Застосування вказівників у більшості випадків дає змогу скоротити текст програми та підвищити ефективність її реалізації, а для цілого ряду задач без вказівникових типів фактично не можна обійтись. Здебільшого вказівники використовують для:

- роботи з масивами та символьними рядками;
- опрацювання даних у динамічній пам'яті;
- створення динамічних структур даних;
- передавання у функції адрес даних для їх опрацювання і/або зміни значень.

Слід пам'ятати, що в разі некоректного застосування вказівники можуть легко стати джерелом помилок, виявити які в програмі достатньо складно. Тому використання вказівників вимагає від програміста особливої уваги.

7.1. Оголошення вказівників, звертання до даних через вказівники

Вказівник (або *покажчик)* – це особливий тип даних, значенням якого є адреса певного байта оперативної пам'яті. Найчастіше вказівник зберігає адресу першого байта одного з об'єктів програми. Це називають *посиланням* на вказаний об'єкт.

Вказівники можуть бути як константними, так і змінними:

- константні вказівники зберігають незмінну адресу певної ділянки оперативної пам'яті, зокрема константними вказівниками є імена масивів і символьних рядків;

- вказівники-змінні є змінними програми, їм можна надавати значення адрес різних ділянок оперативної пам'яті.

Всі вказівники-змінні обов'язково повинні бути оголошені. Для цього застосовують таку синтаксичну конструкцію:

базовий_тип * *ім'я_вказівника*;

тут *базовий_тип* – тип об'єктів програми, адреси яких може зберігати даний вказівник, – це може бути довільний стандартний або користувацький тип; * – ознака того, що наступна змінна є вказівником; *ім'я_вказівника* – звичайний ідентифікатор.

Подамо приклади оголошень вказівників:

```
char *pc;                  /* вказівник на дані з типом char */
unsigned *p1, *p2;         /* два вказівники на дані з типом unsigned */
long *q, h, w;             /* q - вказівник, h і w - звичайні змінні */
```

У наведених прикладах вказівник `pc` може зберігати довільні адреси, які будуть розглядатись як адреси даних з типом `char`; вказівники `p1` та `p2` призначені для збереження адрес даних з типом `unsigned int`; вказівник `q` може посилатись лише на дані, що мають тип `long int`. Знак * в оголошеннях вказівників слід записувати перед кожною вказівниковою змінною, змінні без * вважаються звичайними змінними відповідного типу. Так, в останньому оголошенні вказівником є тільки змінна `q`, а `h` та `w` – звичайні змінні з типом `long int`.

Обсяг ділянки пам'яті, яку займає вказівник,залежить від апаратно-програмної організації збереження даних в оперативній пам'яті і найчастіше становить 2 або 4 байти. У середовищі Borland C розмір вказівника визначається встановленою моделлю пам'яті. Крім цього, вказівники можна явно оголошувати як *ближні* (`near`-вказівники), що займають 2 байти, та *дальні* (`far`- і `huge`-вказівники), що мають розмір 4 байти. Особливості оголошення та використання таких вказівників розглядаються в розділі 14.

З вказівниками пов'язані дві базові операції: `&` – визначення адреси та * – звертання до об'єкта за адресою. Обидві операції унарні. Розглянемо їх.

Операція & – визначення адреси змінної. Значенням даної операції є адреса операнда (змінної), перед яким стоїть знак `&`.

Нехай у програмі оголошено:

```
int  a, *pa;               /* a - змінна,  pa - вказівник */
```

Тоді вираз `&a` буде визначати адресу першого байта ділянки, яку займає змінна `a` в оперативній пам'яті (у Borland C це адреса молодшого з двох байтів, виділених для `a`).

Значення адреси змінної можна присвоїти тим вказівникам, які мають базовий тип, що збігається з типом змінної. Тобто присвоєння

```
pa = &a;                   /* правильне присвоєння */
```

буде цілком коректним, оскільки `a` має тип `int`, а `pa` є вказівником на дані з типом `int`. Водночас два наступні присвоєння:

```
pc = &a;      q = &a;      /* помилкові присвоєння */
```

будуть помилковими, оскільки pc оголошено як вказівник, що посилається на тип char, а q – як вказівник з базовим типом long int.

Встановлено обмеження щодо застосування операції &, а саме:

- не можна визначати адресу константи – записи &150 та &"КІНЕЦЬ" хибні;
- не можна визначати адресу виразу – запис &(a+14) теж хибний;
- не можна визначати адресу змінної з класом пам'яті register (про класи пам'яті даних мова йтиме в розділі 12).

В оголошеннях вказівники можна відразу ініціалізувати значенням адреси об'єкта, що має відповідний до вказівника тип. Наприклад, правильним буде таке оголошення з ініціалізацією:

```
double  z, *pz = &z;                /* ініціалізація адресою змінної */
```

Змінну z у цьому оголошенні записано перед вказівником pz, отже на момент ініціалізації pz для змінної z вже буде виділено ділянку оперативної пам'яті, що має розмір sizeof(double). Вказівник pz отримає адресу першого байта цієї ділянки.

Вказівники можна ініціалізувати також значеннями адрес-констант. У наступному прикладі вказівник adr ініціалізується шістнадцятковою константою, що адресує заданий байт оперативної пам'яті.

```
char *adr=(char*)0x04c0;            /* ініціалізація константною адресою */
```

У заголовному файлі <stdio.h> визначено спеціальну макроконстанту NULL, яку можна присвоювати вказівникам усіх типів (у більшості систем програмування NULL дорівнює 0). Вважається, що вказівник, значенням якого є NULL, не посилається на жоден об'єкт програми, його називають *порожнім* вказівником.

Наведемо приклади використання порожніх вказівників:

```
 int  *start = NULL;                /* початково start неозначений   */
do {                                /* цикл виконується, поки start  */
   . . .                            /* не отримає конкретне значення */
} while (start == NULL );
```

Операція * – звертання за адресою. Операндом операції * є вказівник (адреса ділянки пам'яті), а результатом – значення об'єкта, адресу якого зберігає цей вказівник, – так само, як у разі звертання до об'єкта за його іменем. Цю операцію часто називають *розадресацією*, вона є зворотною до операції взяття адреси &. Результат розадресації має тип, що був заданий як базовий в оголошенні вказівника. Звертання через вказівник можна використовувати всюди, де синтаксично має бути записаний об'єкт даного типу.

Якщо в програмі оголошено змінну і вказівник, що зберігає адресу цієї змінної (так, як це було зроблено вище для z та pz), то в разі присвоєння:

```
*pz = 27.82;                        /* те ж саме, що й  z=27.82; */
```

змінна z набуде значення 27.82. Тобто до z можна звертатись як через її ім'я, так і через вказівник pz, наприклад:

```
if (*pz > MAX)  *pz = MAX;          /* контроль значення z */
```

```
*pz *= 2;                    /* збільшення z у 2 рази:  z = z*2;  */
```

За старшинством операції взяття адреси `&` і розадресації `*` поступаються тільки первинним операціям (див. табл. 4.1), що в більшості випадків знімає потребу використання дужок у виразах з цими операціями.

Оскільки вказівники використовуються в більшості С-програм, а розуміння їх дії і практичне застосування може на початках викликати деякі труднощі, то ще раз підкреслимо, що значенням вказівника є адреса даного, а звертання через вказівник рівносильне звертанню через ім'я змінної. Для нашого останнього прикладу:

- `&z` та `pz` – позначають адресу змінної `z` в оперативній пам'яті;
- `z` та `*pz` – позначають поточне значення змінної `z`.

Наведемо коротку програму, яка наочно демонструє сказане. Для виведення адрес у функції `printf()` використано специфікатор формату `%p`, що записує адреси у формі, яка встановлюється конкретною системою програмування.

```
/*************************************************************/
/*   Звертання до даних через ідентифікатори та вказівники   */
/*************************************************************/
#include <stdio.h>
int main (void)
{
     unsigned  u = 157, *pu = &u;
     long double  w = 0.3631, *pw = &w;
  printf("\n  u=%d          *pu=%d ", u, *pu);
  printf("\n &u=%p      pu=%p      &pu=%p \n", &u, pu, &pu);
  printf("\n  w=%.3Lf        *pw=%.3Lf ", w, *pw);
  printf("\n &w=%p      pw=%p      &pw=%p \n", &w, pw, &pw);
  return 0;
}
```

Приклад виконання програми в середовищі Borland C (адреси виводяться у формі *сегмент* : *зміщення*, обидва значення шістнадцяткові):

```
u = 157          *pu = 157
&u = 9000:0FFE     pu = 9000:0FFE      &pu = 9000:0FFA
 w = 0.363        *pw = 0.363
&w = 9000:0FF0     pw = 9000:0FF0      &pw = 9000:0FEC
```

Застосування кваліфікатора `const`. В оголошеннях вказівників можна вказувати кваліфікатор `const`. Роль цього кваліфікатора визначається місцем його запису.

Якщо слово `const` в оголошенні передує базовому типу вказівника, то воно означає, що такому вказівнику заборонено змінювати значення об'єкта, адресу якого він зберігає. Розглянемо наступні оголошення:

```
int  num,  bak;                  /* звичайні змінні */
const int  cnst=80;              /* константна змінна */
const int  *ptc;                 /* вказівник на константну змінну */
```

Вказівник ptc оголошено як вказівник на константні значення, він може зберігати адресу довільного даного, що має тип int (як константного, так і звичайного):

```
ptc = &cnst;                                    /* правильно */
ptc = &num;       ptc = &bak;                   /* правильно */
```

До даних, на які вказує ptc, можна звертатись операцією *, але змінювати значення цих даних через вказівник ptc не дозволено – це строго контролює компілятор:

```
printf ("Обсяг - %d\n", *ptc);                  /* правильно */
if ( *ptc != 100 )                              /* правильно */
   *ptc = 100;                                  /* ! помилка */
```

Якби в програмі був оголошений звичайний вказівник:

```
int  * pt;                                      /* звичайний вказівник */
```

то присвоєння йому адреси константної змінної було б помилковим:

```
pt = &cnst;                                     /* ! помилка */
pt = &num;        pt = &bak;                    /* правильно */
```

Вказівники на константні дані часто використовують в оголошеннях параметрів функцій. По-перше, це захищає параметри від небажаних випадкових змін їх значень у процесі виконання функції, по-друге, дає змогу передавати в функцію дані, які оголошені в програмі з кваліфікатором const.

Якщо слово const в оголошенні вказівника записати перед його іменем:

```
int * const  cpt = &num;                        /* константний вказівник */
```

то такий вказівник буде константним. В оголошенні константний вказівник треба обов'язково проініціалізувати (у нашому прикладі cpt ініціалізується адресою num) і змінювати в програмі це значення не можна:

```
*cpt = 100;                                     /* правильно */
cpt = &bak;                                     /* ! помилка */
```

Перший оператор звертається до змінної num, на яку посилається cpt, тому він цілком коректний. У другому операторі робиться спроба змінити значення вказівника cpt. Цей вказівник оголошено як константний, тому компілятор зафіксує помилку.

Якщо ж в оголошенні вказівника кваліфікатор const використати двічі:

```
const  int * const  cptc = &cnst;               /* константний вказівник на
                                                   константну змінну */
```

то cptc буде константним вказівником на константне значення з типом int. Його можна використовувати тільки для зчитування даних за адресою, яку цей вказівник отримав після ініціалізації. Записувати щось за цією адресою чи змінювати значення такого вказівника заборонено:

```
if ( *cptc != 100 )                             /* правильно */
   *cptc = 100;                                 /* ! помилка */
cptc = &num;                                    /* ! помилка */
```

7.2. Адресна арифметика

Зручність роботи з вказівниками безпосередньо пов'язана з можливістю виконання над ними ряду операцій, а саме:

- присвоєння;
- порівняння;
- збільшення/зменшення;
- віднімання.

Три останні з перелічених операцій називають *адресною арифметикою*, хоча часто цим терміном позначають усі названі операції над вказівниками.

Операції присвоєння, порівняння та віднімання бінарні, обидва операнди цих операцій обов'язково повинні бути вказівниками (або адресними виразами) з однаковим базовим типом. Операції збільшення чи зменшення значень вказівників можуть бути як унарними (постфіксний або префіксний інкремент/декремент вказівника), так і бінарними – збільшення/зменшення адреси на задану цілочислову величину.

Присвоєння та порівняння вказівників. Вказівнику-змінній можна присвоїти значення адреси, яка зберігається в іншому вказівнику чи формується виразом, заданим справа від знака присвоєння. Необхідною умовою операції присвоєння для вказівників є однаковість базових типів вказівника-змінної і виразу, що присвоюється, інакше потрібно виконувати явне перетворення вказівникового типу правого операнда.

Наприклад, нехай в програмі оголошено:

```
int *pc, *pnew, c = 500;
```

та виконано два присвоєння:

```
pc = &c;          pnew = pc;
```

Тоді обидва вказівники `pc` і `pnew` будуть мати однакове значення – адресу змінної `c`, а значення виразу `*pnew` дорівнюватиме значенню змінної `c` і виразу `*pc`.

Над вказівниками (адресами) можна виконувати всі операції порівняння, що застосовуються у мові С: `<`, `<=`, `>`, `>=`, `==`, `!=`. Результатом кожної операції порівняння у разі її істинності є ненульове значення, а у разі хибності – нуль.

Приклад порівняння значень вказівників:

```
if (pnew == pc)              /* якщо значення вказівників збігаються */
   pnew = NULL;              /* pnew стає невизначеним */
```

Необхідно зазначити, що використання перших чотирьох операцій порівняння: `<`, `<=`, `>` та `>=` доцільне лише в тому випадку, коли обидва вказівники посилаються на елементи одного і того ж масиву. Відповідний приклад наведено далі.

Збільшення/зменшення адрес. До значення вказівника можна додавати (чи від нього віднімати) довільне ціле число. Результатом операції збільшення вказівника `ptr`:

```
ptr = ptr + k;
```

є нова адреса, що формується зі значення, яке зберігає `ptr`, збільшеного на значення

`k*sizeof(`*базовий_тип*`)`. Тут `k` – цілочислова величина, що додається до вказівника, а `sizeof(`*базовий_тип*`)` – розмір об'єкта, на який посилається вказівник `ptr`. Відповідно значенням виразу `ptr-k` є адреса, менша за значення `ptr` на таку ж саму величину. Таким чином, вираз, в якому збільшується або зменшується значення вказівника, формує адресу об'єкта, розташованого на `k` елементів правіше (у разі операції додавання) чи лівіше (операція віднімання) від того об'єкта, на який посилається даний вказівник.

Нехай у програмі оголошено масив із 50-ти довгих цілих чисел і відповідний вказівник:

```
long  mas[50], *pel;
```

Якщо вказівнику `pel` присвоїти адресу першого елемента масиву (у С-програмах він має індекс 0):

```
pel = &mas[0];
```

то вираз `pel+1` буде відповідати адресі наступного елемента масиву, а вираз `pel+5` формує адресу елемента, що має індекс 5. Відповідно вираз `*(pel+1)` – це значення елемента масиву, наступного за `*pel`, а вираз `*(pel+5)` – це значення елемента, зсуненого на п'ять елементів від початку масиву.

Ще раз підкреслимо, що величина, яка додається до значення вказівника чи віднімається від нього, є кратною розміру об'єкта. Якщо, наприклад, значення адреси початку масиву, яке отримав вказівник `pel`, дорівнює `0x1a30`, то значенням виразу `pel+1` є адреса `0x1a34`, а виразу `pel+5` – адреса `0x1a44` (більша на 5×4=20 байтів за адресу, занесену в `pel`).

Операції збільшення/зменшення разом з операцією присвоєння застосовують для зміни значень вказівників. Зокрема, інкремент

```
ptr++;                  /*  або   ptr = ptr+1  */
```

збільшує адресу, занесену в `ptr`, на розмір базового об'єкта. Після виконання цього оператора `ptr` буде вказувати на наступний елемент масиву. Декремент

```
ptr--;                  /*  або   ptr = ptr - 1  */
```

зміщує вказівник `ptr` на попередній об'єкт. Через операції комбінованого присвоєння можна пересувати вказівники відразу на декілька об'єктів. Зокрема, оператор

```
ptr += 10;              /*  або   ptr = ptr + 10  */
```

зсуває `ptr` на 10 елементів вправо, а оператор

```
ptr -= 2*n;             /*  або   ptr = ptr - 2*n  */
```

переносить цей вказівник на `2*n` елементів ліворуч.

Наведені вище операції над вказівниками використано в наступній програмі, яка переставляє елементи заданого масиву цілих чисел у зворотному порядку. Звертання до елементів масиву реалізовано через вказівники `p1` і `p2`. Спочатку `p1` вказує на перший елемент масиву, а `p2` – на останній. У циклі відбувається обмін значеннями елементів, адреси яких задають `p1` і `p2`, а самі вказівники зсуваються назустріч один одному операціями відповідно інкремента та декремента.

```
/*********************************************************/
/*   Переставлення елементів масиву в зворотному порядку   */
/*********************************************************/
#include <stdio.h>
#define  N 10                                  /* кількість елементів масиву*/
int main(void)
{
    int ar[N]={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *p1, *p2;
  p1=&ar[0];                                  /* адреса першого елемента масиву */
  p2=p1+N-1;                                 /* адреса останнього елемента масиву */
          /* цикл переставлення елементів */
  for( ; p1<p2; p1++, p2--)                  /* просуваємо вказівники назустріч */
  {  int d=*p1;                               /* d - додаткова внутрішня змінна */
     *p1=*p2;    *p2=d;                          /* обмін значеннями елементів */
  }
  puts("\n  Масив з переставленими елементами:");
  for( p1=&ar[0]; p1<&ar[N]; p1++ )
     printf("%4d", *p1);                                /* виведення результату */
  return 0;
}
```

У разі виконання програми на екран буде виведено:

```
  Масив з переставленими елементами:
 9   8   7   6   5   4   3   2   1   0
```

Віднімання вказівників. Ця операція виконується над двома вказівниками (адресами), її результатом є кількість елементів базового типу, які можна розташувати в ділянці оперативної пам'яті, адреси початку і кінця якої задають ці вказівники.

Якщо вказівнику `p1` присвоїти адресу першого елемента масиву `ar`, а вказівнику `p2` – адресу шостого елемента (з урахуванням, що індекси починаються з нуля):

```
p1 = &ar[0];          p2 = &ar[5];
```

то значенням різниці `p2-p1` буде число `5` – кількість елементів масиву від першого до шостого.

Інші операції над вказівниками, крім перелічених, не допускаються: не можна додавати чи множити вказівники, виконувати над ними побітові операції тощо.

Характерні вирази з вказівниками. На завершення наведемо деякі характерні вирази та оператори за участю вказівників, що часто використовуються в практичних С-програмах. Їх правильне розуміння і застосування важливе для успішного програмування мовою С.

Нижче записано ряд операторів присвоєння, операндами яких є елементи масиву `mvr`, змінна `a` та вказівники `pm` і `px`. Пропонуємо спробувати самостійно проінтерпретувати записані оператори і визначити, що буде результатом їх виконання, а потім

порівняти свої висновки з наведеними поясненнями. Нехай не лякають вас можливі помилки в інтерпретаціях наведених виразів. Адже вказівники – новий і не зовсім простий у використанні засіб програмування, для вільного володіння яким потрібні як знання, так і певний практичний досвід.

Отже, нехай у програмі оголошено:

```
int mvr[100];          /* масив із 100 цілих чисел */
int *pm = &mvr[0];     /* вказівник на початок масиву  */
int a, *px = pm;       /* цілочислова змінна і додатковий вказівник */
```

Треба вказати результат кожного з наступних операторів:

- `pm++;`

– вказівник пересувається на наступний елемент масиву;

- `a = *pm++;`

– змінна `a` отримує значення елемента, на який вказує `pm`, після чого вказівник пересувається на наступний елемент масиву. Цей результат пов'язаний з тим, що операції постфіксного інкремента та декремента мають вищий пріоритет, ніж операція розадресації, тому знак `++` відноситься до вказівника, а не до виразу `*pm`, хоча сама операція збільшення `pm` виконується останньою, тобто після присвоєння;

- `a=(*pm)++;`

– змінна `a` отримує значення елемента масиву, на який вказує `pm`, після чого цей елемент (але не `a` і не вказівник) збільшується на `1`;

- `a = *++pm;`

– обидві операції `++` та `*` мають однакове старшинство та асоціативність справа наліво, тому спочатку збільшується адреса, записана в `pm`, а потім змінна `a` отримує значення елемента масиву за новою адресою;

- `a = ++*pm;`

– значення елемента масиву, на який вказує `pm`, збільшується на `1`, після чого це нове значення присвоюється змінній `a`;

- `*px++ = *pm++;`

– за адресою `px` записується значення елемента, на який вказує `pm`, а потім обидва вказівники пересуваються на наступні елементи масиву.

7.3. `void`-вказівники. Типізація вказівників

Нетипізовані вказівники. Як вже зазначалось, вказівнику можна присвоювати адреси тільки тих даних, тип яких збігається з базовим типом вказівника. Виняток становлять спеціальні нетипізовані вказівники, оголошені з ключовим словом `void`:

```
void *pv;
```

Якщо вказівник оголошено як нетипізований, то він сумісний з усіма вказівниками програми та адресами усіх об'єктів. Тому вказівнику `pv` можна присвоїти адресу змінної цілого типу `pv=&a` або дійсного типу `pv=&z`, чи будь-якого іншого типу.

Мова С дозволяє присвоювати значення void-вказівників вказівникам з довільним базовим типом і навпаки. Наприклад, якщо в програмі є ще два вказівники:

```
unsigned char  *p1;
long  *p4 = &mas[2];
```

то цілком коректними будуть наступні присвоєння:

```
pv = p4;          p1 = pv;
```

які перепишуть у вказівник p1 адресу, що була записана в p4. Проте, слід пам'ятати, що p1 все одно буде розглядати об'єкт за цією адресою як дане з типом unsigned char, а не long. Тобто звертання *p1 виділить тільки перший байт із long-значення елемента масиву mas, на який посилається p4.

Нетипізовані вказівники широко застосовують для оголошення параметрів функцій, що можуть набувати значення адрес об'єктів різних типів, а також щоб забезпечити можливість повернення з функції адреси неозначеного об'єкта. Наприклад, стандартна бібліотечна функція malloc(), яка здійснює виділення в динамічній пам'яті ділянки заданого обсягу, має тип результату void * (повертає адресу першого байта виділеної ділянки), що дає змогу використовувати її для розташування в динамічній пам'яті об'єктів різних типів.

Ось два приклади присвоєння значення, поверненого функцією malloc(), вказівникам з різними базовими типами:

```
  char *pst;
  double *par
pst = malloc( 300 );
par = malloc( n * sizeof(double) );
```

Після виконання malloc() вказівник pst отримає адресу першого байта ділянки динамічної пам'яті обсягом 300 байтів, а вказівник par – адресу першого байта першого елемента динамічного масиву з n даних, що мають тип double. Тепер можна реалізувати такі присвоєння:

```
*pst = 'X';
*(par+3) = 14.675;
```

У результаті виконання наведених операторів першим символом рядка, на який вказує pst, стане символ 'X', а константу 14.675 буде записано як четвертий за порядком елемент масиву дійсних чисел, адресу початку якого зберігає вказівник par.

Перетворення типу вказівника. Хоча в С-програмах значення вказівника з типом void можна присвоювати довільному іншому вказівникові, все ж у таких операціях програмісти часто застосовують операцію явного перетворення типу. Зокрема, в попередніх прикладах звертання до функції malloc() можна явно перетворити значення, яке повертає malloc(), відповідно до типу вказівника:

```
pst = (char *)malloc( 300 );
par = (double *)malloc( n * sizeof(double) );
```

Явне перетворення типу void-вказівників обов'язкове для мови C++, тому його виконують і в програмах, написаних мовою C, щоб забезпечити їх сумісність з вимогами компілятора C++.

Операцію перетворення типу застосовують також у разі присвоєння вказівникам константних адрес та в разі ініціалізації вказівників константними значеннями:

```
unsigned far *VM = (unsigned far *)0xb8000000;
```

Змінна VM набуває значення адреси першого байта відеопам'яті, в якій зберігається екранне зображення в текстових режимах роботи комп'ютера.

Використовуючи операцію перетворення типу, можна присвоїти вказівнику значення іншого вказівника, що має відмінний базовий тип. Наприклад, для оголошених вище вказівників p1 і p4 можна виконати безпосереднє присвоєння

```
p1 = (unsigned char *)p4;
```

уникнувши застосування нетипізованого вказівника (раніше для такого переприсвоєння використовувався додатковий вказівник pv). Щоб уникнути помилкових результатів роботи програми, треба бути обережним з присвоєнням вказівникам адрес об'єктів іншого типу і застосовувати такі присвоєння тільки там, де вони справді необхідні.

Ще одним випадком використання явного перетворення типу є звертання до даних через void-вказівник. Оскільки такий вказівник не задає розміру об'єкта, адресу якого він зберігає, то для звертання до даних через нетипізований вказівник необхідно перетворити його до відповідного типу.

Проілюструємо сказане прикладом програми, яка виділяє з довгого цілого числа dat, що має розмір 4 байти, значення обох його половин (двобайтових слів) і кожного з чотирьох байтів зокрема. Порядок звертання до складових компонентів dat враховує схему "зворотного" запису цілих чисел в оперативній пам'яті комп'ютера: найменшу адресу має молодший байт числа, а найбільшу – старший (рис. 7.1). Така схема запису цілочислових даних властива, зокрема, комп'ютерам IBM PC.

Рис. 7.1. Схема розташування змінної dat в оперативній пам'яті

Для звертання до всього довгого цілого числа та окремо до його складових частин (слів і байтів) у програмі використано єдиний нетипізований вказівник p. Щоб задати

тип об'єкта, на який в даний момент посилається вказівник p, в усіх звертаннях до даних першою виконується операція перетворення цього вказівника до потрібного типу: (unsigned *) – у разі виділення слів, (unsigned char*) – для виділення байтів та (unsigned long *) – для звертання до всього слова. Значенням виразу *(unsigned*)p є беззнакове двобайтове слово, адресу початку якого задає p, тобто молодша половина dat. Для виділення другої половини dat (старшого слова) спочатку p перетворюється до типу unsigned*. Потім до значення отриманого типізованого вказівника додається 1 – цей вираз формує адресу другого слова. Далі виконується звертання за сформованою адресою:

```
w2 = *((unsigned*)p + 1);
```

Аналогічним способом по черзі виділяються всі чотири байти змінної dat:

```
byte[k] = *((unsigned char*)p + k);
```

```
/*****************************************************/
/*  Виділення слів і байтів із довгого цілого числа  */
/*****************************************************/

#include <stdio.h>
int  main (void)
{
    unsigned long dat = 0x12ab34cd;              /* задане довге число */
    unsigned int w1, w2;                            /* складові слова */
    unsigned char byte[4];                    /* масив складових байтів */
    void *p = &dat;                          /* нетипізований вказівник */
    int k;
  w1 = *(unsigned *)p;                      /* виділення молодшого слова */
  w2 = *((unsigned *)p + 1);                 /* виділення старшого слова */
  for (k = 0;  k < 4;  k++)                    /* цикл виділення байтів */
     byte[k] = *((unsigned char *)p + k);
  printf("\n\tЗадане число - %#0.10lx\n", *(unsigned long *)p);
  printf("Слова:  молодше - %#0.6x \t старше - %#0.6x \n", w1, w2);
  printf("Байти: 1-й (наймолодший) - %#0.4x \t 2-й - %#0.4x \n "
         "\t 3-й - %#0.4x\t4-й (найстарший) - %#0.4x \n",
         byte[0], byte[1], byte[2], byte[3]);
  return 0;
}
```

Результат виконання програми:

```
        Задане число - 0x12ab34cd
Слова:  молодше - 0x34cd       старше - 0x12ab
Байти: 1-й (наймолодший) - 0xcd       2-й - 0x34
       3-й - 0xab       4-й (найстарший) - 0x12
```

Запитання та завдання для самоконтролю

1. Які з оголошених нижче змінних є вказівниками?

```
int  c0, c1, *v, sh;
float  *m1, m2, *rt;
```

2. Які значення можна присвоювати вказівнику?
3. Яким буде значення змінної x після виконання наступного фрагмента програми?

```
int x, *vk;
vk = &x;      x = 5;
*vk = x * 5;      x += *vk;
```

4. Що позначає константа NULL? До яких вказівників її можна застосовувати?
5. Які операції називають операціями адресної арифметики? Які вимоги до вказівників, що вступають у ці операції?
6. Що відбувається у разі виконання операції інкремента/декремента вказівника? Чому дорівнює різниця значень вказівників?
7. Як впливає місце запису кваліфікатора const в оголошенні вказівника на функції цього вказівника?
8. У програмі оголошено два вказівники і змінну n:

```
double  *px;    double* const py = (double*)200;
int n = 0;
```

У процесі виконання даної програми вказівнику px послідовно присвоювались такі значення:

1) `px = py + 50;`
2) `px += 10;`
3) `px = px >= (double*)600 ? px - 10 : NULL;`

Яким буде значення змінної n після виконання наступного оператора?

```
if (px != NULL)
   n = px - py;
```

9. В яких випадках застосовують явне перетворення типу вказівника?
10. У програмі виконано ряд оголошень:

```
char  *pch,    void  *pt;
int  *d, cbr;
```

Які з наведених нижче операторів присвоєння правильні, а які помилкові?

1) `pch = &cbr;`
2) `pt = &cbr;`
3) `d = &cbr;`
4) `pt = pch + cbr;`
5) `pch = (int*)pt;`
6) `cbr *= *d;`
7) `cbr = 3*d;`
8) `cbr = *pt + 8;`
9) `cbr = *(int*)pt - 8;`
10) `*d = cbr**pch;`

Розділ 8

МАСИВИ

У цьому розділі:

- Властивості масивів мови С
- Оголошення та ініціалізація масивів, звертання до елементів масиву через індекси
- Вказівникова форма звертання до елементів масиву
- Багатовимірні масиви: оголошення, розташування в пам'яті, ініціалізація
- Індексна форма звертання до елементів багатовимірного масиву
- Пониження вимірності багатовимірного масиву
- Комбіновані форми звертання до елементів багатовимірного масиву
- Оголошення і використання вказівників на підмасиви багатовимірного масиву

Масиви належать до складених (агрегованих) типів даних. За означенням *масив* – це організована певним чином сукупність даних одного типу (їх називають *елементами* масиву). Зазначимо основні властивості масивів мови С:

- масив займає неперервну ділянку оперативної пам'яті;
- всі елементи масиву мають однаковий тип і спільне ім'я;
- тип елементів масиву може бути довільним простим або складеним;
- елементи масиву розташовуються послідовно за порядком зростання індексів;
- індекс першого елемента масиву завжди дорівнює 0;
- до елементів масиву можна звертатись як через індекси, так і через вказівники;
- ім'я масиву є константним вказівником на його початок в оперативній пам'яті, тобто ім'я масиву зберігає адресу першого елемента цього масиву;
- масив може бути одновимірним чи багатовимірним (масив, сформований з масивів).

Стандарт С-99 дещо розширив способи створення й використання масивів у програмах, зокрема затвердив використання т. зв. складених літералів (див. Додаток 3).

8.1. Одновимірні масиви

Одновимірний масив – це лінійна послідовність однотипних елементів. Такі масиви часто називають математичним терміном *вектор*. Кожен елемент одновимірного масиву має свій порядковий номер (*індекс*), який визначає розташування цього елемента.

8.1.1. Оголошення та ініціалізація масивів

Як і звичайні змінні, всі масиви повинні бути оголошені явно, щоб компілятор міг виділити для кожного з них ділянку пам'яті відповідного обсягу. Оголошення масивів виконують через таку синтаксичну конструкцію:

тип_елементів ім'я_масиву [*кількість_елементів*];

тут *тип_елементів* – довільний допустимий для С простий чи складений тип; *ім'я_масиву* – ідентифікатор, що відповідає правилам запису імен; квадратні дужки [] – обов'язкова ознака масиву; *кількість_елементів* – константа чи константний вираз, що визначає *розмірність* даного масиву.

Наведемо приклади оголошень масивів:

```
double  arr[15];            /* масив з 15-ти дійсних чисел */
int  vector[4*N];           /* масив з 4*N цілих чисел, N - константа */
```

Для кожного масиву компілятор виділяє неперервну ділянку пам'яті, обсяг якої дорівнює добутку *кількість_елементів*×`sizeof`(*тип_елементів*). Наприклад, масив `arr` буде займати 15×8=120 байтів, а його елементи матимуть індекси від `0` до `14`. В оголошенні масиву `vector` кількість елементів задається виразом, в якому `N` має бути попередньо визначеною макроконстантою чи константною змінною (хоча деякі компілятори С не дозволяють використовувати в оголошеннях масивів змінні з кваліфікатором `const`).

Елементи масиву завжди розташовуються в оперативній пам'яті послідовно один за одним. Найменшу адресу має перший елемент (тобто елемент з індексом `0`), а найстаршу – останній елемент масиву (рис. 8.1).

Рис. 8.1. Порядок розташування елементів масиву `arr`

Оголошуючи масиви, можна відразу ініціалізувати їх елементи. Застосовують дві форми ініціалізації:

- із зазначенням кількості елементів масиву;
- без зазначення кількості елементів масиву.

У першому випадку масив оголошується звичайним чином, після чого у фігурних дужках `{}` послідовно вказуються константні значення елементів масиву, починаючи від першого (з індексом `0`). Наприклад:

```
int simple1[5] = {2, 3, 5, 7, 11};
```

Кількість заданих значень не повинна перевищувати вказану розмірність масиву, але може бути меншою за неї – тоді ініціалізуються тільки початкові елементи масиву.

У разі оголошення:

```
int simple2[40] = {2, 3, 5};
```

значення отримають тільки перші три зі сорока елементів масиву `simple2`, решта елементів залишаться невизначеними (або дорівнюватимуть нулю, якщо масив оголошено як глобальний чи статичний, – ці питання детальніше розглядаються в розділі 12).

Якщо в оголошенні масиву відбувається його повна ініціалізація, то можна опустити значення *кількість_елементів*. У цьому випадку розмірність масиву буде визначатись кількістю елементів-ініціалізаторів. Наприклад:

```
char golosni[] = {'a', 'o', 'e', 'u', 'i', 'y'};
```

Масив `golosni` отримає ділянку для шести елементів з типом `char`, кожен з елементів відразу буде заповнений кодом відповідного символа. Квадратні дужки `[]` обов'язкові для всіх форм оголошення масиву.

Розмірність масиву, оголошеного без вказання кількості елементів, можна визначити в програмі, використовуючи операцію `sizeof`:

кількість_елементів = `sizeof(`*ім'я_масиву*`) / sizeof(`*тип_елементів*`)`

Зокрема, наступний вираз обчислює кількість елементів масиву `golosni`:

```
sizeof(golosni) / sizeof(char)
```

Операція `sizeof(`*ім'я_масиву*`)` повертає розмір (обсяг у байтах) усієї ділянки, яку займає масив, незалежно від того, наскільки він заповнений. Тобто значенням виразу `sizeof(arr)` буде число `120`, навіть якщо в цей масив ще не заносились елементи взагалі, а значення `sizeof(simple2)` дорівнюватиме `80` (40 двобайтових елементів) у всій програмі, хоча початково проініціалізовано тільки три перші елементи даного масиву.

8.1.2. Звертання до елементів масиву через індекси та через вказівники

Елемент масиву є змінною, яка має тип, заданий в оголошенні масиву, і синтаксично може використовуватись у програмі всюди, де може бути записане дане такого типу.

Для доступу до елементів масиву використовуються дві форми звертання:

- через індекси;
- через вказівники (адреси).

Індексне звертання. У разі звертання до елементів масиву через індекси застосовують конструкцію

ім'я_масиву `[`*індекс_елемента*`]`

де *індекс_елемента* може бути довільним виразом, який має значення цілого типу і задає порядковий (індексний) номер елемента в масиві (нагадаємо, що нумерація індексів починається з `0`).

Подамо приклади індексних звертань:

`simple2[0]` – перший елемент масиву `simple2`, його значення дорівнює 2;

`golosni[4]` – п'ятий за порядком елемент масиву `golosni`, його значенням є символ `'i'`;

`arr[2*k-5]` – елемент масиву `arr`, індекс якого дорівнює значенню виразу `2*k-5`.

Треба пам'ятати, що компілятор мови С не контролює відповідність значень індексів до розмірності масиву.

У разі виконання оператора

```
num = arr[16];
```

змінна `num` отримає значення двобайтового слова, розташованого за адресою, де мав би зберігатись сімнадцятий елемент масиву `arr`. Оскільки `arr` за оголошенням містить тільки 15 елементів, то значення, присвоєне `num`, буде випадковим. Оператор

```
golosn[10] = 'ю';
```

запише код літери `'ю'` в байт, який є одинадцятим від початку масиву `golosn`. Зрозуміло, що такі некоректні звертання можуть призвести до серйозних помилок у роботі програми і навіть збою системи. Тому в критичних точках програми доцільно виконувати перевірку правильності індексів, наприклад, так:

```
if (i>=0 && i<15)
   arr[i] = ...;
```

У мові С квадратні дужки `[]`, в яких записують індекс елемента масиву, є знаком операції, що за пріоритетом належить до найстаршого рівня і має асоціативність зліва направо. Ці властивості індексного звертання визначають порядок обчислення виразів, серед операндів яких є елементи масивів.

Проілюструємо індексну форму доступу до елементів масиву прикладом програми, яка створює новий масив на основі заданого масиву дійсних чисел. Кожен елемент нового масиву дорівнює півсумі сусідніх елементів базового. Додаткові пояснення наведемо після програми.

```
/*************************************************************/
/*   Формування нового масиву, кожен елемент якого дорівнює  */
/*   півсумі сусідніх елементів базового масиву              */
/*************************************************************/
#include <stdio.h>
int  main(void)
{
  double  base[] = {0.34, 14.68, 123.8, 21.78, 90.07, 55.6, 12.07};
  const int  nsize = sizeof(base)/sizeof(double)-1;
  double  nar[nsize];          /* новий масив, що має розмірність nsize */
  int i;
   for (i = 0;  i < nsize;  i++)                /* формування масиву nar */
      nar[i] = (base[i]+base[i+1])/2;
```

```
    printf("\n\t\t  Новий масив: \n");
    for (i=0;  i<nsize; i++)                    /* виведення нового масиву */
       printf("%7.2lf", nar[i]);
    return 0;
}
```

Результат виконання:

```
            Новий масив:
7.54   69.23   72.79   55.93   72.83   33.84
```

Масив `base` у програмі оголошено без вказання кількості елементів – його розмірність визначається кількістю констант ініціалізації. Розмірність нового масиву (вона буде на 1 меншою за розмірність `base`) обчислюється програмно і записується в константну змінну `nsize` (модифікатор `const` в оголошенні `nsize` обов'язковий):

```
nsize = sizeof(base)/sizeof(double)-1;
```

Як вже зазначалось, деякі системи програмування не підтримують використання константних змінних в оголошеннях масивів. У цьому випадку можна виділити для нового масиву необхідну за обсягом ділянку в динамічній пам'яті або просто оголосити розмірність `nar` як константу, значення якої буде не меншим, ніж максимально можлива кількість елементів у новоствореному масиві.

Вся робота з елементами масивів `base` і `nar` здійснюється у програмі через індекси. Звернемо увагу на той факт, що індекс останнього елемента масиву завжди на одиницю менший за кількість елементів даного масиву.

Загалом індексна форма звертання до елементів масиву проста й наочна, оскільки вона явно відображає номер елемента, що опрацьовується.

Адресне звертання. Звертання до елементів масиву через вказівники здебільшого дає змогу створити менший за обсягом та більш ефективний код програми.

Визначальна особливість масивів мови С полягає в тому, що ім'я масиву є константним вказівником на його перший елемент. Для масиву `base` з попереднього прикладу можна записати такі пари рівнозначних виразів (позначимо рівнозначність символом ≡):

```
base ≡ &base[0]                 /* адреса елемента base з індексом 0 */
*base ≡ base[0]                 /* значення першого елемента base */
```

Відповідно, враховуючи операції адресної арифметики:

```
base+i ≡ &base[i]               /* адреса i-го елемента base */
*(base+i) ≡ base[i]             /* значення i-го елемента base */
```

За результатами звертання до елементів масиву обидві форми (індексна та вказівникова) рівнозначні. Проте звертання через адресу елемента відповідає механізмові доступу до елементів масиву, тому кожне індексне звертання перетворюється компілятором у вказівникове. Тобто, вираз `base[i+1]` компілятор перетворює у форму `*(base+i+1)`, а вираз `nar[i]` замінює виразом `*(nar+i)`.

Для роботи з масивами найчастіше використовують додаткові вказівники, в які заносять адреси тих елементів, що опрацьовуються в даний момент. Це забезпечує найвищу швидкодію звертання до елементів масивів.

Для прикладу розглянемо програму, яка визначає значення і номер найбільшого елемента масиву довгих цілих чисел.

```
/*************************************************************/
/*  Визначення номера та значення найбільшого елемента масиву  */
/*************************************************************/
#include <stdio.h>
#define  N 50                          /* максимальна розмірність масиву */
int  main (void)
{
    long arr[N];                     /* оголошення масиву з N елементів */
    long *pel,                         /* вказівник на поточний елемент */
         *pend,                          /* вказівник на кінець масиву */
         *pmax;            /* вказівник на максимальний елемент масиву */
    int k = 0;                          /* кількість введених елементів */
  printf("\nВведіть елементи масиву, кінець введення - 0\n");
  while(k < N) {                                  /* цикл введення даних */
     scanf("%ld", arr+k);
     if ( *(arr+k) == 0 )                                   /* введено 0 */
        break;
     k++;
  }
  pmax = arr;               /* приймаємо перший елемент за максимальний */
  pend = arr + k;                                 /* адреса кінця масиву */
  for( pel= arr +1; pel < pend; pel++ )                /* цикл перевірки */
      if ( *pel > *pmax )                              /* всіх елементів */
         pmax = pel;          /* фіксуємо адресу максимального елемента */
  printf("Найбільшим є %d-й елемент масиву, його значення - %ld\n",
          (int)(pmax-arr+1), *pmax);
  return 0;
}
```

Приклад виконання:

```
Введіть елементи масиву, кінець введення - 0
788  37500  456  42864  567  3452  870  22703  965  0
Найбільшим є 4-й елемент масиву, його значення - 42864
```

Пояснимо використані в програмі вирази з операндами-вказівниками:

`arr+k` – адреса `k`-го елемента масиву `arr`, те ж саме, що й `&arr[k]`;

`*(arr+k)` – звертання до `k`-го елемента масиву `arr`, те ж саме, що й `arr[k]`;

`pmax = arr` – присвоєння вказівнику `pmax` адреси першого елемента масиву `arr`, те ж саме, що й `pmax = &arr[0]`;

`pend = arr + k` – присвоєння вказівнику `pend` адреси кінця масиву `arr` (тобто адресу k-го елемента, який вже не належить до даних, що опрацьовуються), те ж саме, що й `pend = &arr[k];`

`pel = arr + 1` – присвоєння вказівнику `pel` адреси другого елемента масиву, те ж саме, що й `pel = &arr[1];`

`pel++` – пересування вказівника `pel` на наступний елемент масиву;

`*pel > *pmax` – порівняння елементів масиву, на які вказують `pel` та `pmax`;

`pmax = pel` – запис адреси елемента, на який вказує `pel`, у вказівник `pmax`;

`pmax-arr+1` – порядковий номер максимального елемента масиву, який на 1 більший за різницю адрес вказівників `pmax` і `arr` (залежно від встановленої моделі пам'яті ця різниця може мати тип `int` або `long`).

Ще раз нагадаємо, що ім'я масиву є вказівником-константою, тому над ним не можна виконувати операції присвоєння, зокрема збільшувати чи зменшувати його значення. Операції: `arr=pel`, `arr++` чи `arr+=2` - хибні.

8.2. Багатовимірні масиви

Мова С інтерпретує *багатовимірний масив* як масив масивів, тобто масив, елементами якого є масиви меншої вимірності (*підмасиви*).

8.2.1. Розташування в пам'яті та ініціалізація

Матриця цілих чисел, яку оголошено наступним чином:

```
int matr[10][5];
```

є масивом з десяти елементів, кожен з яких в свою чергу є одновимірним масивом (вектором) з п'яти елементів, що мають тип `int`. Це означає, що матриця `matr` складається з десяти рядків, кожен з яких містить п'ять цілочислових елементів.

Для збереження `matr` в оперативній пам'яті буде виділено неперервну ділянку обсягом `10×5×sizeof(int)` байт, тобто 100 байтів, якщо тип `int` в даній системі програмування є двобайтовим. Схему розташування елементів матриці в оперативній пам'яті показано на рис. 8.2.

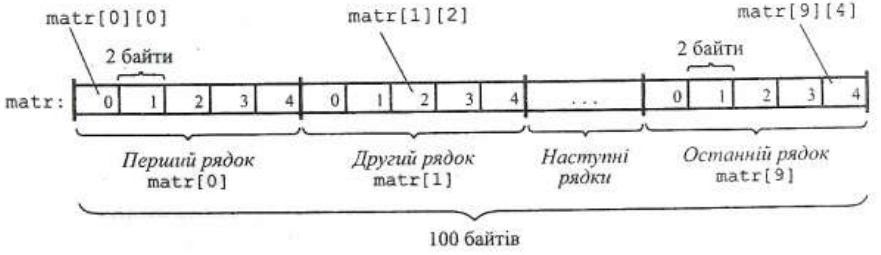


Рис. 8.2. Розташування в пам'яті двовимірного масиву цілих чисел `matr[10][5]`

В оголошеннях багатовимірних масивів вказується в окремих квадратних дужках [] кількість елементів для кожного з вимірів. Спрощені форми оголошення не підтримуються.

Ось приклад оголошення тривимірного масиву:

```
unsigned  form3d [8][60][60];
```

Масив form3d складається з восьми елементів, кожен з яких є матрицею розмірністю 60 на 60 елементів, сформованою з даних, що мають тип unsigned int. Цей масив буде займати ділянку оперативної пам'яті обсягом 8×60×60×sizeof(unsigned) байт, що у випадку двобайтових слів становить 57600 байтів. Працюючи з багатовимірними масивами, треба завжди звертати увагу на їх сумарний обсяг – він не повинен перевищувати обсяг сегмента оперативної пам'яті (64 Кбайт).

Як і одновимірні, багатовимірні масиви можна ініціалізувати відразу в оголошеннях. Наприклад, у разі оголошення:

```
int  m[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

всі елементи матриці m (вона складається з трьох рядків, по три елементи в кожному) будуть заповнені вказаною послідовністю натуральних чисел.

В оголошеннях з ініціалізацією для багатовимірних масивів дозволено опускати розмірність найстаршого виміру (але тільки найстаршого). Попереднє оголошення можна записати ще й так:

```
int  m[ ][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

У цьому випадку кількість рядків матриці обчислюється як частка від ділення кількості даних у списку ініціалізації на відому кількість елементів в одному рядку матриці.

Якщо треба проініціалізувати не всі елементи багатовимірного масиву, а тільки початкові елементи окремих рядків, то застосовують внутрішні роздільчі фігурні дужки {}. У разі наступного оголошення:

```
unsigned  mat[4][6] = {{0, 1, 2}, {5, 11}, {17}};
```

буде сформовано матрицю з чотирьох рядків, по шість цілих беззнакових елементів у кожному. Початкові значення отримають перші три елементи першого рядка матриці mat, два перших елементи другого рядка та один елемент третього, всі інші елементи залишаться непроініціалізованими.

Для доступу до елементів багатовимірного масиву можна застосовувати класичну індексну форму звертання, вказавши в окремих квадратних дужках [] значення індекса для кожного з вимірів.

Наведемо декілька прикладів:

`m[0][2]` – третій елемент першого рядка матриці m;

`mat[i][0]` – перший елемент i-го рядка (рахуючи від 0) матриці mat;

`form3d[k+1][59][58]` – передостанній елемент (k+1)-ї матриці (двовимірного масиву) із тривимірного масиву form3d (59 – індекс рядка, а 58 – індекс елемента в цьому рядку).

Наведена нижче програма визначає суму елементів кожного рядка заданої матриці дійсних чисел. Реальну розмірність матриці (в оголошенні матриці задано її граничні розміри) та значення елементів користувач вводить з клавіатури. Звертання до елементів матриці виконуються в програмі через індексну форму.

```
/*************************************************************/
/*  Обчислення суми елементів кожного рядка введеної матриці */
/*************************************************************/
#include <stdio.h>
#define MAX 10          /* максимальна розмірність */
int main(void)
{
    double mt[MAX][MAX];     /* матриця, розмірністю MAX * MAX */
    int nr, nc,              /* реальна кількість рядків і стовпців */
        k, j;                /* поточні індекси */
    double sum;              /* сума елементів рядка */
  printf("\nРозмірність: ");
  scanf("%d%d", &nr, &nc);
  printf("Елементи матриці [%d*%d]:\n", nr, nc);
  for(k=0; k<nr; k++)           /* цикл введення елементів матриці */
     for(j=0; j<nc; j++)
         scanf("%lf", &mt[k][j]);
  printf("Суми елементів рядків:\n");
   /* цикл обчислення суми елементів кожного рядка матриці */
  for(k=0; k<nr; k++) {
    for(j=0, sum=0; j<nc; j++)
       sum += mt[k][j];              /* сумування елементів k-го рядка */
    printf("%2d-го: %8.2lf\n", k+1, sum);
  }
  return 0;
}
```

Приклад виконання:

```
Розмірність: 4  5
Елементи матриці [4*5]:
9.25  7.81  14.32  8.5  13
18.43  16  19.07  14  28.75
11.7  19.4  26.9  42.06  7.4
27.1  14.88  19.2  6.3  2.41
Суми елементів рядків:
 1-го:    52.88
 2-го:    96.25
 3-го:   107.46
 4-го:    67.89
```

8.2.2. Вказівники у багатовимірних масивах

Правильне використання вказівників у програмах, що опрацьовують багатовимірні масиви, вимагає чіткого розуміння особливостей адресації та звертання до складових частин цих масивів.

Пониження вимірності масиву. Як вже зазначалось, всі елементи багатовимірного масиву займають спільну неперервну ділянку оперативної пам'яті (див. рис. 8.2), адресу початку якої зберігає константний вказівник – ім'я масиву. Тому багатовимірний масив можна розглядати як одновимірний, в якому кожен елемент має свій порядковий номер. Приміром, елемент `matr[k][j]` матриці, зображеної на рис. 8.2, буде мати номер `k*5+j`, де `5` – кількість елементів у рядку матриці; а останній елемент цієї матриці буде мати номер `49` (нумерацію ведемо, починаючи з `0`).

Хоча ім'я багатовимірного масиву й зберігає адресу початку (першого байта) цього масиву, проте слід пам'ятати, що за своїм змістом ім'я багатовимірного масиву є вказівником на перший підмасив даного масиву, а не на його перший елемент. Тому щоб отримати адресу першого елемента масиву, треба виконати операцію перетворення імені масиву до типу вказівника на елементи цього масиву. Вираз `(int*)matr` буде визначати адресу першого елемента матриці `matr`:

```
(int*)matr ≡ &matr[0][0]     /* адреса першого елемента matr */
```

а вираз

```
*(int*)matr ≡ matr[0][0]     /* значення першого елемента matr */
```

задає значення цього елемента. Відповідно:

```
(int*)matr+k*5+j ≡ &matr[k][j]      /* адреса елемента matr[k][j] */
*((int*)matr+k*5+j) ≡ matr[k][j]    /* значення цього елемента */
```

Нагадаємо, що операції перетворення типу (*тип*) та розадресації `*` мають однаково високий рівень старшинства й асоціативність справа наліво. У виразі `*(int*)matr` спочатку створюється вказівник з типом `int *`, який задає адресу першого елемента масиву, а вже потім виконується звертання до цього елемента.

Пониження вимірності імені багатовимірного масиву до рівня вказівника на елементи цього масиву є зручним прийомом, коли треба опрацьовувати повністю заповнений масив. Найкраще для цієї мети ввести додатковий зовнішній вказівник і присвоїти йому адресу першого елемента масиву:

```
int *ptr = (int*)matr;
```

Тепер можна ефективно просуватись по всіх наступних елементах матриці `matr`, інкрементуючи значення вказівника `ptr`.

Подана далі програма використовує описаний прийом для визначення кількості парних елементів у заданій матриці цілих беззнакових чисел.

```
/*****************************************************/
/*   Визначення кількості парних елементів матриці   */
/*****************************************************/
#include <stdio.h>
#define NC 4                              /* кількість елементів у рядку */
int main (void)
{
    unsigned mat[][NC]={{5, 6, 7, 5}, {2, 3, 4, 8}, {9, 1, 3, 4}};
    unsigned *pmxel;                  /* вказівник на елементи матриці */
    int nel,                      /* загальна кількість елементів матриці */
        keven=0;                        /* кількість парних елементів */
  nel = sizeof(mat)/sizeof(unsigned);
      /* цикл по всіх елементах матриці */
  for (pmxel=(unsigned*)mat; pmxel<(unsigned*)mat+nel; pmxel++)
      if (*pmxel%2 == 0)  keven++;                  /* елемент парний */
  printf("\n Парних елементів - %d. \n", keven);
  return 0;
}
```

Результат виконання:

```
Парних елементів - 5.
```

Завдяки вказівнику `pmxel` звертання до елементів матриці таке ж просте й швидке, як і звертання до елементів одновимірного масиву. Проте такий спосіб опрацювання багатовимірного масиву можна застосовувати тільки за умови, що цей масив повністю заповнений. Інакше після опрацювання необхідної кількості елементів поточного рядка (чи підмасиву) треба певним чином переставити вказівник, який адресує елементи багатовимірного масиву, на початок наступного рядка (або підмасиву).

Комбіновані звертання до елементів масиву. Інший підхід базується на тому, що багатовимірний масив є масивом, який складається із набору підмасивів меншої вимірності. Зокрема, вираз `matr[0]` або `*matr` виділяє перший елемент масиву `matr`, який в прикладі, проілюстрованому на рис. 8.2, є одновимірним масивом з п'яти елементів, що мають тип `int`. За означенням ім'я кожного масиву С-програми – це константний вказівник на його перший елемент, тому:

```
matr[0] ≡ *matr ≡ &matr[0][0]          /* масив першого рядка матриці */
```

Так само значенням виразів `matr[k]` і `*(matr+k)` є масив, що збігається з k-м рядком матриці `matr`:

```
matr[k] ≡ *(matr+k) ≡ &matr[k][0]                  /* масив k-го рядка */
```

Послідовність значень `matr[0]`, `matr[1]`, ... `matr[9]` формує віртуальний (уявний) масив вказівників-констант, кожна з яких є адресою початку (тобто першого елемента) відповідного підмасиву (рядка) матриці `matr` (рис. 8.3). У прикладі, поданому на рис. 8.3, матриця займає неперервну ділянку оперативної пам'яті, яка починається

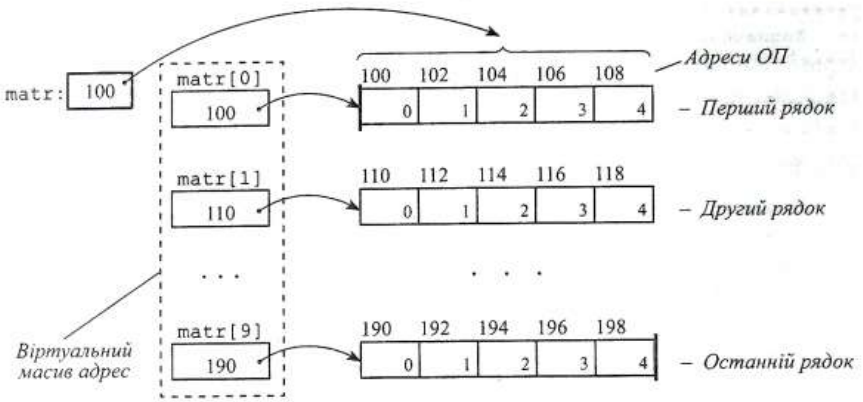


Рис. 8.3. Адресація двовимірного масиву `matr[10][5]`

байтом за адресою 100 (перший байт елемента `matr[0][0]`) і закінчується байтом, що має адресу 199 (останній байт елемента `matr[9][4]`).

Значення адреси, яка зберігається в `matr`, і значення `matr[0]` однакові – це адреса початку масиву. Але треба пам'ятати, що між цими двома вказівниками є принципова відмінність: `matr[0]` вказує на перший елемент першого рядка, а `matr` вказує на цілий перший рядок даної матриці (див. рис. 8.3).

Оскільки вираз `matr[k]` є константною адресою першого елемента k-го рядка, то вираз `matr[k]+j` обчислює адресу `j`-го елемента цього рядка:

```
matr[k]+j ≡ *( matr+k)+j ≡ &matr[k][j]      /* адреса j-го елемента */
```

До елементів багатовимірного масиву можна звертатись, використовуючи тільки індексну чи тільки вказівникову форму доступу, а також різні комбінації обох форм. Наведемо декілька характерних прикладів, які найчастіше зустрічаються в програмах:

- звертання до першого елемента матриці:

```
matr[0][0] ≡ *matr[0] ≡ **matr
```

- звертання до `j`-го елемента першого рядка матриці:

```
matr[0][j] ≡ *(matr[0]+j) ≡ *(*matr+j)
```

- звертання до першого елемента k-го рядка матриці:

```
matr[k][0] ≡ *matr[k] ≡ **(matr+k)
```

- звертання до `j`-го елемента k-го рядка матриці:

```
matr[k][j] ≡ *(matr[k]+j) ≡ (*(matr+k))[j] ≡ *(*(matr+k)+j)
```

У загальному випадку звертання до елемента *n*-вимірного масиву вимагає обов'язкового виконання *n* операцій розадресації, які можуть бути зроблені через індексну і/або вказівникову форму звертання.

Застосування вказівників на підмасиви. Якщо для роботи з багатовимірними масивами використовують додаткові зовнішні вказівники, то вони можуть бути оголошені як вказівники безпосередньо на елементи масиву чи як вказівники на можливі базові підмасиви даного багатовимірного масиву. Застосування кожного типу вказівників має свої особливості в організації звертання до елементів багатовимірного масиву.

Наприклад, вказівнику, який оголошено так:

```
int *pel;
```

можна присвоїти адресу довільного елемента матриці `matr`, оскільки його базовий тип збігається з типом, який мають елементи матриці:

```
pel=*matr;                  /* адреса елемента matr[0][0] */
```

чи

```
pel=*(matr+2)+4;            /* адреса елемента matr[2][4] */
```

Відповідно оператор

```
*pel=50;
```

запише число `50` за адресою, занесеною в `pel` останньою, тобто в елемент матриці `matr[2][4]`.

Вказівник, сумісний з матрицею `matr`, треба оголосити інакше:

```
int (*pm)[5];               /* вказівник на масив з п'яти int-елементів */
```

Круглі дужки `()` в оголошенні вказівника на масив необхідні, оскільки операція `[]` має вище старшинство, ніж операція `*`. Без дужок `()` це було б оголошення масиву з п'яти вказівників на дані, що мають тип `int`:

```
int *m[5];                  /* масив з п'яти вказівників на int-дані */
```

Щоб вказівник був сумісним з типом багатовимірного масиву, він повинен бути вказівником на базовий підмасив цього масиву.

Оголошеному вище вказівнику `pm` можна присвоювати адресу довільного масиву, що складається з п'яти елементів, які мають тип `int`. Тому `pm` може слугувати вказівником на кожен з рядків матриці `matr`. Якщо виконати присвоєння

```
pm=matr;                    /* вказівник на масив 0-го рядка */
```

то значення вказівника `pm` дорівнюватиме адресі першого рядка матриці `matr`. Тоді результатом звертання `*pm` буде масив, який збігається з цим рядком матриці:

```
*pm ≡ *matr ≡ matr[0] ≡ &matr[0][0]          /* масив 0-го рядка */
```

Оскільки масив у мові С задається адресою свого першого елемента, то масив, що відповідає першому рядку матриці, буде визначатись адресою елемента `matr[0][0]`: `matr[0] ≡ &matr[0][0]`.

Відповідно, якщо виконати присвоєння:

```
pm=matr+6;                  /* вказівних на масив 6-го рядка */
```

то pm буде вказувати на масив, що є сьомим рядком (рахуючи від першого) матриці matr, а звертання *pm:

```
*pm ≡ *(matr+6) ≡ matr[6] ≡ &matr[6][0]      /* масив 6-го рядка */
```

виділяє весь цей масив і є еквівалентним до адреси його першого елемента. Загалом, *pm ≡ &matr[k][0], де k – номер рядка, на який вказує pm.

Вираз **pm – це звертання до першого елемента одновимірного масиву (тобто рядка матриці), адресу якого задає підвираз *pm:

```
**pm ≡ **(matr+k) ≡ *matr[k] ≡ matr[k][0]   /* 0-й елемент рядка k */
```

Відповідно вирази

```
*(*pm+j) ≡ (*pm)[j] ≡ *(*(matr+k)+j) ≡ (*(matr+k))[j] ≡
≡ *(matr[k]+j) ≡ matr[k][j]                  /* j-й елемент рядка k */
```

визначають j-й елемент рядка, на який вказує pm.

Наведена далі програма реалізує введення матриці дійсних чисел, що має розмірність nr × nc, та виконує перетворення введеної матриці шляхом ділення елементів кожного рядка (крім першого) на значення відповідних елементів першого рядка. У програмі продемонстровано використання різних способів і форм доступу до рядків та елементів матриці. Додаткові пояснення зроблено після програми.

```
/*****************************************************************/
/*  Різні форми вказівникового звертання до елементів матриці  */
/*****************************************************************/
#include <stdio.h>
#define N 15
int main (void)
{
    double arr[N][N];
    double (*prow)[N],        /* вказівник на рядки матриці */
           *p0el,             /* вказівник на елементи 0-го рядка */
           *pirel;            /* вказівник на елементи i-го рядка */
    int nr, nc, i, j;
  printf("\nРозмірність: ");
  scanf("%d%d", &nr, &nc);
  printf("Елементи матриці (%d*%d):\n", nr, nc);
  for (j=0, p0el=arr[0]; j<nc; j++, p0el++)
      scanf("%lf", p0el);          /* введення елементів першого рядка */
  /* цикл введення елементів усіх наступних рядків і
     ділення їх на відповідні елементи першого рядка */
  for (i=1; i<nr; i++)
    for (p0el=*arr, pirel=*(arr+i); p0el<*arr+nc; p0el++, pirel++){
       scanf("%lf", pirel);
       *pirel /= *p0el;
    }
  printf("\n  Перетворена матриця:\n");
```

```
  /* цикл виведення матриці зі зміненими елементами */
  for (prow=arr;  prow<arr+nr;  prow++){
     for (j=0;  j<nc;  j++)
        printf("%8.2f", *(*prow+j));    /* виведення елементів рядка */
     printf("\n");
  }
  return 0;
}
```

Приклад виконання:

```
Розмірність: 3 4
Елементи матриці (3*4):
9.25    7.81   14.32   8.5
11.7   19.4    26.9   42.06
27.1   14.88   19.2    6.3

   Перетворена матриця:
  9.25    7.81   14.32    8.50
  1.25    2.48    1.88    4.95
  2.93    1.91    1.34    0.74
```

Для звертання до елементів першого (що має індекс 0) рядка матриці `arr` у програмі використано вказівник `p0el`, який вказує на дані з типом `double`. Початково `p0el` отримує адресу першого елемента цього рядка `p0el=arr[0]`, а далі операцією інкремента `p0el++` послідовно просувається по всіх інших елементах цього рядка.

У циклі введення та ділення елементів наступних рядків матриці використано ще один вказівник на елементи масиву – `pirel`. Цикл опрацювання елементів кожного рядка починається з ініціалізації вказівників `p0el` та `pirel`: перший встановлюється на початок нульового рядка матриці `p0el=*arr`, а другий – на початок `i`-го рядка `pirel=*(arr+i)`. Введене за адресою `pirel` значення ділиться на значення відповідного за номером елемента першого рядка:

```
*pirel /= *p0el;
```

Після цього обидва вказівники пересуваються на наступні елементи своїх рядків.

Для виведення елементів матриці використано вказівник `prow`, який оголошено як вказівник на масив з `N` елементів, що мають тип `double`:

```
double (*prow)[N];
```

Після ініціалізації `prow = arr` вказівник `prow` набуває адреси першого рядка матриці, тобто значенням виразу `*prow` стає масив, що збігається з першим рядком. Кожне збільшення вказівника `prow++` пересуває `prow` на наступний рядок матриці. Вираз `*(*prow+j)` реалізує звертання до `j`-го елемента поточного рядка, на який вказує `prow`. Цей вираз можна було б записати і через індексну форму звертання до елементів масиву: `(*prow)[j]`. Дужки в останньому виразі необхідні, щоб першою виконувалась операція розадресації `*prow` – її результатом є масив елементів поточного рядка, а вже потім – операція `[j]`, яка виділяє `j`-й елемент цього рядка.

Запитання та завдання для самоконтролю

1. Назвіть визначальні властивості масивів, встановлені в мові С.
2. Як оголошуються одновимірні масиви? В яких випадках можна опускати параметр *кількість_елементів* в оголошеннях масивів?
3. Порівняйте індексну і вказівникову форми звертання до елементів одновимірного масиву. Які переваги кожної з цих форм?
4. У програмі оголошено та проініціалізовано масив і відповідний вказівник:

   ```
   double vec[10]={1.28, -7.4, -3.22, 6.74, 8.14}, *pv=vec+4;
   ```

   Вкажіть значення кожного з наступних виразів:

   1) `vec[0]>vec[1]` 2) `*vec+2` 3) `*(vec+2)`
   4) `*pv*2` 5) `2**pv` 6) `*--pv`
5. Як оголошуються та ініціалізуються багатовимірні масиви?
6. У програмі оголошено двовимірний масив цілих чисел:

   ```
   int ar2d[][6]={{5, 12, -7}, {-3, 6, 3, 8}, {14, 4, -6, 2}};
   ```

   Вкажіть значення кожного з наступних виразів:

   1) `ar2d[2][0]` 2) `*(ar2d[1]+2)` 3) `**ar2d`
   4) `*ar2d[1]` 5) `(*ar2d)[1]` 6) `*(*(ar2d+1)+3)`
7. Оголосіть вказівник на рядки масиву `ar2d` з п.6.

Запрограмуйте наведені нижче задачі, використовуючи різні форми звертання до елементів масиву: • на основі індексів • через розадресоване ім'я масиву та зміщення елемента • за допомогою допоміжних вказівників

8. Використовуючи бібліотечні функції генерації рівномірно розподілених випадкових чисел, сформувати і вивести на екран масив з `N` цілих випадкових чисел, що потрапляють у діапазон `1..M`. Всі елементи масиву повинні бути різними.
9. Оголосити й проініціалізувати масив довгих цілих чисел. Циклічно зсунути його елементи на один уліво. Надрукувати результат зсування. *Підказка*: циклічне зсування передбачає, що перший елемент масиву переноситься на місце останнього.
10. З клавіатури ввести масив дійсних чисел, упорядкований за зростанням значень елементів (розмірність масиву в оголошенні повинна перевищувати кількість введених чисел). Вставити в масив заданий додатковий елемент так, щоб уся послідовність чисел залишалась упорядкованою. Вивести на екран доповнений масив.
11. З клавіатури ввести прямокутну матрицю цілих беззнакових чисел. Утворити одновимірний масив з номерів рядків матриці, які складаються з тих самих чисел, що й перший рядок (порядок запису чисел у рядку може бути довільним). Надрукувати сформований масив або вивести повідомлення про відсутність відповідних рядків.
12. Для заданої квадратної матриці дійсних чисел знайти та надрукувати найбільший квадрат. Найбільшим квадратом вважається довільна квадратна частина заданої матриці, сума елементів якої максимальна (це, зокрема, може бути вся початкова матриця або тільки якийсь один її елемент).

Розділ 9

СИМВОЛЬНІ РЯДКИ

У цьому розділі:

- Символьні рядки як особливий вид масиву, оголошення та ініціалізація символьних рядків
- Звертання до символів рядка через індекси та через вказівники
- Функції введення і виведення символів
- Введення/виведення символьних рядків: функції `gets()` та `puts()`, особливості форматного введення та виведення рядків
- Стандартні бібліотечні функції класифікації та перетворення символів
- Функції `<string.h>`, призначені для опрацювання символьних рядків: копіювання, доповнення і порівняння рядків, визначення довжини рядка, виділення лексем та інші
- Функції перетворення символьних рядків у числа та зворотних перетворень
- Масиви символьних рядків і масиви вказівників на перші символи рядків
- Запис символьних рядків у динамічну пам'ять, переваги використання динамічної пам'яті для збереження й опрацювання символьних рядків

Мова С не має спеціального типу для оголошення *символьних рядків* (інші назви: *рядки символів* або *стрінги*), а розглядає символьний рядок як особливий вид масиву. Елементи масиву, який називають символьним рядком, мають тип `char`, їх значеннями є коди символів, з яких складається цей рядок (ASCII-коди, якщо дана система програмування застосовує ASCII-таблицю для кодування символів). Останнім символом рядка повинен бути т. зв. нуль-символ (`'\0'`), код якого дорівнює `0`. З кожним символьним рядком пов'язується вказівник на початок даного рядка. У всьому іншому символьні рядки повністю зберігають властивості масивів.

9.1. Оголошення та ініціалізація символьних рядків

Пригадаємо, що рядкові константи в мові С записуються як охоплена лапками послідовність довільних символів: `"..."`. В оперативній пам'яті їм виділяється ділянка, обсяг якої на один байт більший за кількість символів у рядку. В цей додатковий байт автоматично записується нуль-символ, який надалі слугуватиме ознакою кінця рядка.

Реально рядкові константи не є повноправними константами С. Стандарт розглядає рядкову константу як статичний масив, проініціалізований заданими символами. Реакція на спробу змінити рядкову константу визначається особливостями конкретної системи програмування.

Синтаксично рядкова константа рівнозначна вказівнику на перший символ свого рядка, тому вона повністю сумісна з типами `const char*` та `char*`.

Нехай у програмі оголошено:

```
char * pst = "Це символьний рядок";
```

Для збереження записаного рядка компілятор виділить 20 байтів, з них 19 байтів для символів і останній для `'\0'` (рис. 9.1). Адресу початку рядка отримає вказівник `pst`.

Символьні рядки, які є змінними програми, оголошуються як звичайні масиви:

```
char ім'я_символьного_рядка [кількість_символів];
```

Оголошений нижче масив `str` призначений для збереження символьного рядка:

```
char str[150];
```

У `str` можна записати довільний символьний рядок, довжина якого не перевищує 149 символів, оскільки останнім записується нуль-символ – для нього треба обов'язково зарезервувати один байт. Слід також пам'ятати, що перевищення встановленої в оголошенні кількості символів не контролюється компілятором і може призвести до небезпечних помилок у роботі програми.

В оголошеннях символьні рядки, як і масиви символів, можна ініціалізувати. Розглянемо декілька характерних прикладів:

```
char m1[20] = {'a', 'b', 'c', 'd', 'e', 'f'};
char m2[20] = {'a', 'b', 'c', 'd', 'e', 'f', '\0'};
char m3[20] = "abcdef";
char m4[ ] = "abcdef";
```

Символьні рядки `m1`, `m2` і `m3` оголошено однаково – як масиви з 20 елементів, що мають тип `char`, але ініціалізацію їх виконано різними способами. Початкові шість елементів масиву `m1` заповнено послідовністю літер, проте без `'\0'` у кінці, тому цей масив не буде повноправним символьним рядком, з ним можна буде працювати тільки як зі звичайним масивом символів. У масив `m2` записано таку саму послідовність літер, а після неї – нуль-символ. Фактично в `m2` занесено рядок символів `"abcdef"`. Такий самий рядок записано в масив `m3`, тобто результати ініціалізації `m2` та `m3` збігаються (очевидно, що ініціалізація `m3` є простішою у записі та сприйнятті). Незаповнені

Рис. 9.1. Розташування в пам'яті константного символьного рядка

елементи масивів m1, m2 та m3 містять "сміття" (за умови, що масиви оголошено як локальні, а в разі глобальних чи статичних масивів усі вільні елементи заповнюються нулями). Надлишкові елементи можна використовувати надалі для доповнення і розширення відповідних рядків. В оголошенні масиву m4 не вказано граничну кількість символів, тому розмірність цього масиву (символьного рядка) встановлюється за кількістю елементів-ініціалізаторів. Для наведеного прикладу розмірність m4 становитиме 7 символів: 6 перших байтів масиву заповнюються кодами літер, а сьомий – кінцевим нуль-символом. Хоча масиви m3 та m4 проініціалізовані константними рядками, елементи цих масивів можна змінювати так само, як елементи масивів m1 та m2.

9.2. Звертання до елементів символьних рядків

Процеси опрацювання символьних рядків у С-програмах базуються на двох основних властивостях рядків:

1) ім'я символьного рядка є константним вказівником на його перший символ;
2) кінець рядка задається нуль-символом '\0'.

Для звертання до символів рядка застосовують як індексну, так і вказівникову форму доступу до елементів масиву, які були детально розглянуті в попередньому розділі. Проілюструємо це прикладом двох варіантів програми, яка вилучає зі символьного рядка всі входження заданого символа. У першому варіанті використано індексне звертання до елементів рядка, а в другому – вказівникове.

```
/*****************************************/
/*   Витирання заданого символа. Варіант 1   */
/*****************************************/
#include <stdio.h>
int  main (void)
{
    char  st[] = "ABC *** XYZ *** KM*Q**RT*";       /* заданий рядок */
    char  sym = '*';                  /* символ, що має бути вилучений */
    int  k,n;                                      /* індекси символів */
  k = n = 0;
  while (st[k] != '\0')               /* поки не досягнуто кінця рядка */
     if (st[k] == sym)                    /* знайдено потрібний символ */
        k++;                           /* пропуск символа без перезапису */
     else {
       st[n] = st[k];                  /* копіювання всіх інших символів */
       n++;  k++;
     }
  st[n] = '\0';                       /* фіксація кінця стиснутого рядка */
  printf ("\nРядок без символа %c:  %s\n", sym, st);
  return 0;
}
```

Результат виконання:

```
Рядок без символа *:  ABC  XYZ  KMQRT
```

У наступному варіанті програми для звертання до елементів символьного рядка введено вказівники pk і pn. Перший вказує на символ, який перевіряється, а другий – на позицію рядка, куди повинен бути переписаний символ, що залишається в рядку. Замість циклу while у другому варіанті програми використано цикл for, а умову оператора if змінено на протилежну.

```
/*****************************************/
/*  Витирання заданого символа. Варіант 2  */
/*****************************************/
#include <stdio.h>
int  main(void)
{
    char  st[] = "ABC *** XYZ *** KM*Q**RT*";        /* заданий рядок */
    char  sym = '*';                  /* символ, що має бути вилучений */
    char  *pk, *pn;                     /* вказівники на символи рядка */
  for (pk=pn=st;  *pk != '\0';  pk++)       /* цикл по символах рядка */
    if (*pk != sym)
      *pn++ = *pk;           /* копіювання всіх символів, крім заданого */
  *pn = '\0';                             /* фіксація кінця нового рядка */
  printf("\nРядок після вилучення:  %s\n", st);
  return 0;
}
```

Результат виконання:

```
Рядок після вилучення:  ABC  XYZ  KMQRT
```

У циклі for даної програми умовний оператор:

```
if (*pk != sym)
   *pn++ = *pk;
```

послідовно перевіряє кожен символ рядка st, звертаючись до елементів через вказівник pk. Якщо поточний символ не збігається з тим, який задано для вилучення (*pk != sym), то він переписується у позицію, на яку вказує pn (*pn=*pk). Після цього pn пересувається на наступний символ рядка (pn++).

9.3. Введення/виведення символів і символьних рядків

Бібліотеки мови С містять набір функцій, призначених для введення та виведення одиночних символів і цілих символьних рядків.

Функції введення/виведення, прототипи яких оголошено в заголовному файлі <stdio.h>, належать до стандартних функцій мови – вони є в усіх реалізаціях С. Ці функції здійснюють потокоорієнтоване буферизоване введення/виведення даних. У процесі введення набрана на клавіатурі інформація заноситься у спеціальний внутрішній буфер і одночасно відображається на екрані. Вона передається у програму для

опрацювання тільки після того, як натиснено клавішу *Enter*. Це дає змогу виправляти помилкові символи та вносити зміни в символьний рядок перед тим, як він надійде на опрацювання.

Додаткові можливості надають функції консольного введення/виведення, які оголошено в заголовному файлі `<conio.h>`. Вони не належать до групи функцій, що підтримуються стандартом мови С, але такі чи подібні функції є в бібліотеках більшості систем програмування. Функції консольного введення не виконують буферизації даних, тобто дані передаються в програму відразу після їх введення, а в разі введення одиночного символа – відразу після натискання на клавішу цього символа. Функції консольного виведення забезпечують керування формою екранного зображення, зокрема через них можна встановлювати позицію виведення і колір екранних символів. Детальний опис цих функцій подано в розділі 16.

Зараз розглянемо тільки функції буферизованого введення та виведення символів і символьних рядків. Крім універсальних функцій форматного введення та виведення `scanf()` і `printf()`, про які мова вже йшла раніше, бібліотека С містить ряд функцій, призначених спеціально для введення/виведення одного символа та цілого символьного рядка.

9.3.1. Введення/виведення символів

Основною функцією буферизованого введення одного символа з клавіатури (точніше зі стандартного потоку `stdin`) є функція `getchar()`, яку оголошено так:

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

Функція `getchar()` не має параметрів і повертає код зчитаного символа, доповнений до типу `int` старшим нульовим байтом. У разі виявлення помилки введення функція повертає макроконстанту `EOF`. Ця макроконстанта також оголошена в `<stdio.h>`, для більшості систем програмування вона має значення `-1`.

Виведення одного символа виконує функція

```
int  putchar (int sym);
```

Параметром `putchar()` є код символа, що має бути відображений на екрані (або виведений у стандартний потік `stdout`). За умови успішного завершення функція повертає код виведеного символа, а в разі невдачі – макроконстанту `EOF`.

Проілюструємо використання описаних функцій прикладом програми, яка відображає на екрані введений з клавіатури символьний рядок, доповнюючи кожен символ короткою рискою (дефісом).

```
/*********************************************/
/*  Демонстрація функцій getchar() і putchar()  */
/*********************************************/
#include <stdio.h>
int  main (void)
{
```

```
    int sym;
    printf("\nРядок:  ");
    while ((sym=getchar())!='\n') {      /* цикл зчитування символів */
       putchar(sym);                     /* виведення символа */
       putchar('-');                     /* виведення риски */
    }
    putchar('\b');                       /* повернення на один символ */
    putchar(' ');                        /* витирання останньої риски */
    return 0;
}
```

Приклад виконання:

```
Рядок:  12345abcdXYZ
1-2-3-4-5-a-b-c-d-X-Y-Z
```

Звернемо увагу на декілька моментів. У виразі умови оператора `while`:

```
(sym=getchar())!='\n'
```

реалізовано відразу три дії: 1 – функція `getchar()` вводить один символ рядка; 2 – код введеного символа присвоюється змінній `sym`; 3 – виконується перевірка, чи введений символ є кодом клавіші *Enter* (`'\n'`). Операція `!=` за старшинством є вищою, ніж операція присвоєння `=`, тому вираз `sym=getchar()` необхідно взяти в дужки. Щоб витерти риску, виведену після останнього символа, у кінці програми виводиться керуючий символ `'\b'` – "повернення на крок", а за ним символ пробілу `' '`. У процесі реалізації програми вхідний рядок буде зображено на екрані двічі: перший раз під час набору його на клавіатурі та занесення символів у буфер введення, а другий раз – у процесі зчитування символів з буфера і виведення їх на екран з проміжною рискою.

Змінна `sym`, що використовується у програмі для введення і друку символів, має тип `int`. Це зумовлене тим, що функція `getchar()` повертає цілочисловий результат, а `putchar()` має аргумент `int`-типу. Проте реально з двобайтового цілого використовується тільки значення молодшого байта, тому змінну `sym` можна було оголосити і як змінну з типом `char`. Загалом функції символьного введення/виведення можна застосовувати для роботи з даними, що мають тип `int` або `char`.

Щоб виконувати форматне введення або виведення символів за допомогою функцій `scanf()` і `printf()`, у рядку формату вказують специфікацію `%c`. У разі введення символа відповідний елемент списку введення `scanf()` повинен бути адресою, за якою буде записано зчитаний символ. Для виведення одного символа у списку параметрів `printf()` треба вказати вираз, значення якого має тип `int` або `char` і задає код символа (як і у випадку функції `putchar()`, дані з типом `char` доповнюються до типу `int`, але до уваги береться тільки молодший байт `int`-значення).

9.3.2. Введення/виведення символьних рядків

Введення символьного рядка. Введення рядка символів виконує функція

```
char * gets (char * st);
```

Ця функція послідовно зчитує всі символи, введені з клавіатури (у загальному випадку – зі стандартного потоку введення `stdin`), і записує їх у масив, адреса початку якого задається параметром-вказівником `st`. Ознакою кінця введення слугує символ нового рядка `'\n'`, який заноситься у буфер введення при натисканні на *Enter*. Замість символа `'\n'` функція записує у рядок кінцевий нуль-символ `'\0'`.

У разі успішного завершення `gets()` повертає вказівник на початок введеного рядка, тобто адресу першого символа рядка `&st[0]`, а в разі виникнення помилки – порожній вказівник `NULL`.

Дуже важливо, щоб розмір масиву `st` був достатнім для введення всіх символів рядка. Функція `gets()` не контролює кількість зчитаних символів, тому в разі переповнення `st` результат роботи програми буде непередбачуваним.

Запишемо коротку програму, в якій зчитується символьний рядок, а потім підраховується його довжина.

```
/*****************************************/
/*  Визначення довжини символьного рядка  */
/*****************************************/
#include <stdio.h>
int  main (void)
{
     char  buf[150], *p=buf;
   printf ("\nРядок:  ");
   gets (buf);
   while (*p)                              /* тобто поки *p != '\0' */
      p++;                                 /* пошук кінця рядка */
   printf("\nДовжина рядка - %d символів.\n", (int)(p-buf));
   return 0;
}
```

Приклад виконання:

```
Рядок:  Рахуємо кількість символів.
Довжина рядка - 27 символів.
```

Кількість символів у введеному рядку визначено через різницю значень вказівників `p-buf`: вказівник `buf` містить адресу першого символа введеного рядка, а вказівник `p` вказує на кінцевий нуль-символ цього рядка.

Введення символьного рядка часто поєднують з перевіркою або присвоєнням значення, яке повертає функція `gets()`. Це робить програму лаконічнішою. Наприклад, наступний оператор, що перевіряє перший символ введеного рядка:

```
if( *gets(buf) == 'A' )
   . . .                                 /* відповідні дії */
```

виконує дію двох операторів:

```
gets(buf);
if( buf[0] == 'A')
   . . .                                 /* необхідні дії */
```

Виведення символьного рядка. Виведення рядка символів здійснює функція

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

Ця функція має один параметр `st` – вказівник на початок рядка, який має бути виведений на екран. За умови успішного виконання функція повертає невід'ємне значення, а в разі помилки – макроконстанту `EOF`.

Рядок, що виводиться, повинен обов'язково закінчуватись нуль-символом, інакше після символів рядка будуть виводитись у формі символів значення всіх наступних байтів оперативної пам'яті, поки не зустрінеться байт зі значенням 0. Сам нуль-символ `'\0'` функція замінює символом нового рядка `'\n'`, тобто після виведення всіх символів заданого стрінга екранний курсор завжди переводиться на початок нового рядка.

Виклик функції `puts()` з параметром, що задає порожній рядок:

```
puts("");
```

часто використовують для встановлення курсора на початок наступного екранного рядка в процесі виведення даних.

У наведеній далі програмі реалізовано виділення слів із введеного речення; кожне слово виводиться в окремому рядку. Програма використовує два вказівники: `pw` встановлюється на початок поточного слова, а `ps` – на його кінець. Щоб слова виводились як окремі стрінги, символ пробілу після кожного слова замінюється нуль-символом (будемо вважати, що слова в реченні відокремлені тільки одним символом пробілу).

```
/*****************************************************/
/*  Формування стовпчика слів із речення. Варіант 1  */
/*****************************************************/
#include <stdio.h>
int  main (void)
{
     char  sent[140];              /* масив для введеного речення */
     char  *pw = sent, *ps = sent;
   puts ("\n\tРечення:");
   gets (sent);
   puts ("\n  Слова:");
   while (*ps != '\0')             /* цикл по всіх символах речення */
      if (*ps != ' ')              /* якщо не кінець слова  */
         ps++;                     /* просування по слову   */
      else {                       /* знайдено кінець слова */
         *ps = '\0';               /* фіксація кінця слова  */
         puts (pw);                /* виведення слова */
         pw = ++ps;                /* перехід на наступне слово  */
      }
   puts (pw);                      /* виведення останнього слова */
   return 0;
}
```

Приклад виконання:

```
    Речення:
Формування стовпчика слів <Enter>
  Слова:
Формування
стовпчика
слів
```

Форматне введення/виведення символьних рядків. У разі форматного введення або виведення символьних рядків за допомогою функцій `scanf()` та `printf()` відповідний елемент рядка формату цих функцій повинен бути специфікацією `%s`.

Особливість форматного введення стрінгів через функцію `scanf()` полягає в тому, що з буфера введення зчитується тільки початкова частина символьного рядка до першого символа-роздільника (пробіла, табуляції, нового рядка тощо). Символи-роздільники, які передують першому непробільному символу, опускаються, а кінцем введення слугує поява довільного роздільника після звичайних символів. Тому рядок, введений за допомогою `scanf()`, не буде містити символів пробілу. У ряді випадків це може бути корисним, зокрема через `scanf()` легко організувати зчитування окремих слів із введеного речення. У кінець введеного рядка функція записує `'\0'`.

Щоб не допустити введення надмірної кількості символів, доцільно в специфікації формату задати граничну ширину поля введення, наприклад:

```
  char  newst[30];
scanf("%25s", newst);
```

Тепер з вхідного рядка в `newst` буде занесено не більше, ніж 25 символів.

Функція `printf()` виводить на екран всі символи заданого рядка до нуль-символа. Сам `'\0'` не виводиться і не замінюється іншим символом, тому курсор залишається за останнім виведеним символом.

Проілюструємо роботу функцій `scanf()` та `printf()` ще одним прикладом програми попередньої задачі: роздрукувати в стовпчик слова введеного речення.

```
/*****************************************************/
/*  Формування стовпчика слів із речення. Варіант 2  */
/*****************************************************/
#include <stdio.h>
int  main (void)
{
     char  wrd[20];                  /* масив для введеного слова */
     int  n = 0;
   puts ("\n\tРечення:");
   do {                              /* цикл введення/виведення слів */
      scanf("%19s", wrd);
      printf("Слово %d - %s \n", ++n, wrd);
   } while (getchar()!= '\n');
   return 0;
}
```

Приклад виконання:

```
    Речення:
<Tab> Формування      стовпчика   слів <Enter>
Слово 1 - Формування
Слово 2 - стовпчика
Слово 3 - слів
```

Друга програма не тільки простіша та коротша, але й дає змогу опрацьовувати речення, в яких між словами записано довільну кількість пробільних символів. Символи пробілу і/або табуляції можуть стояти також перед першим символом.

9.4. Бібліотечні функції для роботи з символами та символьними рядками

Задачі аналізу й перетворення текстової інформації належать до найбільш поширених задач комп'ютерних інформаційних технологій. Стандартна бібліотека мови С включає набір різнотипних функцій, що забезпечують швидку реалізацію операцій, які найчастіше зустрічаються у процесах опрацювання символьних і текстових даних.

9.4.1. Функції класифікації та перетворення символів

У заголовному файлі `<ctype.h>` оголошено групу функцій, призначених для перевірки та класифікації окремих символів. Імена цих функцій починаються префіксом `is`: `is...()`. Усі функції мають один параметр з типом `int` – символ, що перевіряється. Функції перевірки повертають ціле ненульове значення (істина), якщо заданий символ належить до відповідної класифікаційної групи, і нульове значення (хибність), якщо символ не належить до цієї групи. Наприклад, функцію, яка перевіряє, чи заданий символ `sym` є літерою (до уваги беруться тільки великі та малі латинські літери), оголошено так:

```
int  isalpha (int sym);
```

У табл. 9.1 наведено найбільш популярні функції класифікації та зміни символів. Повну таблицю функцій `<ctype.h>` подано в табл. Д2.2 Додатка 2.

Дві останні функції з табл. 9.1: `tolower()` та `toupper()` призначені для зміни регістра символів-літер (на жаль, тільки латинських). Перша з цих функцій повертає відповідну малу літеру, якщо `sym` велика латинська літера, а друга – повертає велику літеру, якщо `sym` мала латинська літера. В інших випадках обидві функції повертають значення `sym`.

Роботу описаних функцій перевірки та перетворення символів ілюструє програма, яка посимвольно зчитує рядок із буфера введення до появи першого цифрового символа, а потім друкує введений стрінг великими літерами.

Таблиця 9.1

Основні функції класифікації та зміни символів

Функція	Призначення
класифікації:	Перевіряє, чи символ sym є:
`int isalpha(sym)`	малою або великою латинською літерою
`int isdigit(sym)`	десятковою цифрою
`int isalnum(sym)`	латинською літерою або десятковою цифрою
`int isxdigit(sym)`	шістнадцятковою цифрою
`int isspace(sym)`	пробільним символом (символом пробілу, нового рядка, горизонтальної чи вертикальної табуляції)
`int islower(sym)`	малою латинською літерою
`int isupper(sym)`	великою латинською літерою
перетворення:	Повертає:
`int tolower(sym)`	малу латинську літеру, якщо sym велика латинська літера
`int toupper(sym)`	велику латинську літеру, якщо sym мала латинська літера

```
/*************************************************/
/*  Посимвольне введення і перетворення рядка  */
/*************************************************/
#include <stdio.h>
#include <ctype.h>
#define LEN 100                    /* максимальна розмірність рядка */
int main (void)
{
    char  buf[LEN],                /* буфер введення */
          *pb = buf;               /* вказівник на символи рядка */
   printf("\nРядок: ");
   do {
      *pb=getchar();
      if (isdigit(*pb) || *pb=='\n')          /* зустрілась цифра або */
         break;                               /* кінець рядка */
   } while (++pb<buf+LEN-1);                  /* заповнено весь рядок */
   *pb='\0';
   printf( "\nРезультат:  " );
   for (pb=buf; *pb!='\0'; pb++)              /* цикл виведення рядка */
      putchar(toupper(*pb));
   return 0;
}
```

Приклад виконання:

```
Рядок: The code number is 924567
Результат:  THE CODE NUMBER IS
```

9.4.2. Функції операцій над символьними рядками

Прототипи бібліотечних функцій, призначених для роботи з символьними рядками, оголошені в заголовному файлі <string.h>. Основну групу складають функції, які починаються префіксом str: str...(). У табл. 9.2 описано функції, які найчастіше використовуються у процесах опрацювання символьних рядків (повний набір функцій <string.h> бібліотеки Borland C подано в табл. Д2.3 Додатка 2). Щоб скоротити записи прототипів функцій у табл. 9.2, в списку параметрів кожної функції вказано тільки імена параметрів. Типи параметрів в оголошеннях цих функцій є такими:

```
const  char *s, *s1, *s2;
char  *sr;
unsigned  n;
int  sym;
```

Символьні рядки, які опрацьовуються функціями <string.h>, мають обов'язково закінчуватись '\0'. У разі звертання до функцій конкатенації (об'єднання) та копіювання рядків треба забезпечити, щоб розмірність масиву символів sr була достатньою для запису рядка результату, оскільки функції не контролюють довжин рядків. Крім цього, рядки sr і s у функціях конкатенації та копіювання не повинні перекриватись (у С-99 це зазначається кваліфікатором restrict в оголошеннях параметрів sr і s – див. Додаток 3).

Проілюструємо використання деяких із описаних функцій на прикладі програми, яка вилучає зі символьного рядка підрядок, обмежений зліва і справа символами '#'.

```
/*****************************************************/
/*  Вилучення підрядка зі заданого символьного рядка  */
/*****************************************************/
#include <stdio.h>
#include <string.h>
int  main (void)
{
     char  str[]= "abcdef # 12345679 # uvwxyz";              /* рядок */
     char  *p1, *p2;        /* вказівники на початок і кінець підрядка */
   p1 = strchr(str,'#');                              /* перший символ '#' */
   p2 = strrchr(str,'#');                           /* останній символ '#' */
   strcpy(p1,p2+1);              /* перенесення кінцевої частини рядка */
   printf("\nРядок після вилучення:  %s", str);
   return 0;
}
```

Результат виконання:

```
Рядок після вилучення:  abcdef  uvwxyz
```

Щоб вилучити підрядок, знаходимо його початок (перший символ '#' у str) і кінець (останній символ '#'). Потім переносимо, використовуючи функцію strcpy(), кінцеву частину основного рядка на місце підрядка, який треба витерти.

Таблиця 9.2

Основні функції опрацювання символьних рядків

Функція	Призначення
`char * strcpy (sr, s);`	Копіює рядок `s` (з `'\0'` включно) за адресою, заданою параметром `sr`. Повертає значення `sr` – адресу скопійованого рядка.
`char * strcat (sr, s);`	Долучає рядок `s` (з `'\0'` включно) у кінець рядка `sr`. Повертає значення `sr` – адресу доповненого рядка.
`int strcmp (s1, s2);`	Послідовно порівнює символи рядків `s1` і `s2` як дані з типом `unsigned char`. Повертає ціле число, значення якого: < 0, якщо `s1 < s2`; 0, якщо `s1 == s2`; > 0, якщо `s1 > s2`.
`char * strncpy (sr, s, n);`	Аналог `strcpy()`, але з `s` копіюється не більше, ніж `n` початкових символів; якщо скопійована група символів не закінчується `'\0'`, то нуль-символ у `sr` не заноситься.
`char * strncat (sr, s, n);`	Аналог `strcat()`, але долучає до `sr` тільки `n` початкових символів з `s`; у кінець об'єднаного рядка заноситься `'\0'`.
`char * strncmp (s1, s2, n);`	Аналог `strcmp()`, але порівнює тільки `n` початкових символів рядків `s1` та `s2`. Якщо якийсь із рядків коротший за `n`, то порівняння припиняється з досягненням `'\0'`.
`unsigned strlen (s);`	Повертає довжину рядка `s` у символах (`'\0'` не враховується).
`char * strchr (s, sym);`	Перевіряє, чи символ `sym` входить у рядок `s`. Повертає вказівник на перше входження `sym` у `s` або `NULL`, якщо `sym` не зустрічається в рядку `s`.
`char * strrchr (s, sym);`	Аналог `strchr()`. Повертає вказівник на останнє входження символа `sym` у рядок `s` або `NULL`, якщо `sym` не зустрічається в рядку `s`.
`char * strstr (s1, s2);`	Перевіряє, чи рядок `s2` входить як підрядок у `s1`. Повертає вказівник на перший символ рядка `s2` у `s1` або `NULL`, якщо рядок `s2` не зустрічається у рядку `s1`.
`char * strtok (sr, s);`	Виділяє в рядку `sr` лексеми, обмежені символами з рядка `s` (детальніший опис наведено далі). Повертає вказівник на виділену лексему або `NULL`.
`char * strdup (s);`	Копіює рядок `s` (з `'\0'` включно) в динамічну пам'ять, попередньо виділивши там ділянку потрібної довжини. Повертає адресу рядка в динамічній пам'яті.

Останньою в табл. 9.2 записана функція `strdup()`, яка заносить копію заданого символьного рядка (з нуль-символом включно) у динамічну пам'ять та повертає адресу, за якою записано рядок. Функція `strdup()` не належить до функцій, що підтримуються стандартом мови С, хоча вона входить до складу більшості С-бібліотек. Приклад використання `strdup()` наведено в параграфі 9.5.3.

Функція виділення лексем. Зупинимось детальніше на функції `strtok()`, оголошення якої наступне:

```
char *  strtok (char * str, const char * lim);
```

Ця функція виконує поділ символьного рядка `str` на окремі лексеми, записуючи після кожної лексеми `'\0'`. Рядок `lim` задає набір символів, якими можуть бути обмежені лексеми рядка `str` (нуль-символ у переліку обмежувачів вказувати не треба). Для виділення всіх лексем символьного рядка функцію використовують циклічно. У першому звертанні до `strtok()` вказують адресу початку рядка, а функція повертає адресу першої знайденої лексеми. У наступних звертаннях до `strtok()` замість першого параметра записують порожній вказівник `NULL`, а функція повертає адресу наступної лексеми рядка. Коли всі лексеми виділені, функція повертає `NULL`.

Раніше ми вже записували програму, яка виділяє слова введеного речення і виводить їх на екран у стовпчик. Тоді передбачалось, що слова в реченні відокремлені тільки символом пробілу. За допомогою `strtok()` можна легко розширити можливості програми, доповнивши пробіл іншими роздільниками слів: комою, крапкою, дужками, рискою тощо.

```
/*********************************************/
/*  Виділення слів-лексем із символьного рядка  */
/*********************************************/

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

int  main (void)
{
    char  example[] = "Символи, рядки (виділення слів-лексем)";
    const  char *limits = " ,.;()-";      /* символи-обмежувачі лексем */
    char  *pw;                                  /* вказівник на лексеми */
  printf("\n  Слова: \n");
  pw = strtok(example, limits);          /* знаходження першої лексеми */
  while (pw != NULL) {
     puts(pw);
     pw = strtok(NULL, limits);             /* пошук наступної лексеми */
  }
  return 0;
}
```

Результат виконання:

```
  Слова:
Символи
рядки
виділення
слів
лексем
```

9.4.3. Функції перетворення рядків символів у числа та зворотних перетворень

Перетворення "символьний рядок ⇒ число". У багатьох задачах необхідно перетворювати числові дані, записані у формі текстових рядків, в одну з внутрішніх форм збереження чисел. Такі перетворення реалізують стандартні бібліотечні функції, оголошені в <stdlib.h> (табл. 9.3). Типи параметрів цих функцій наступні:

```
const  char *st;
char  **end;
int  base;
```

Таблиця 9.3

Функції перетворення символьних рядків у числа

Функція	Призначення
`int atoi (st);`	Виділяє у рядку `st` перше ціле десяткове число і перетворює його у дане з типом `int`. Числу може передувати довільна кількість символів пробілу. Кінцем рядка вважається перший символ, що не належить до цифр. Повертає знайдене числове значення у разі успішного перетворення, а в разі помилки – результат не визначений.
`long atol (st);`	Аналог `atoi()`, але перетворює рядок у число з типом `long`.
`double atof (st);`	Аналог `atoi()`, але перетворює рядок `st` у дане з типом `double`. Число у рядку може бути записане як ціле чи як дійсне у формі з фіксованою або з плаваючою крапкою.
`long strtol (st, end, base);`	Розширений варіант `atol()`. Вказівник `end` (цей параметр є вказівником на вказівник) задає адресу змінної-вказівника, в яку буде записано адресу першого символа, що залишився неперетвореним. Параметр `base` визначає основу системи числення, в якій записано число і може приймати значення від 2 до 36: цифрами числа можуть бути арабські цифри і послідовні малі або великі латинські літери (для `base`>10). Якщо `base` дорівнює `0`, то основа числа визначається формою його запису: число, що починається `0`, вважається вісімковим, число з префіксом `0X` чи `0x` – шістнадцятковим, всі інші числа розглядаються як десяткові.
`unsigned long strtoul (st, end, base);`	Аналог `strtol()`, але повертає значення, що має тип `unsigned long`.
`double strtod (st, end);`	Розширений варіант `atof()`. Перетворює початкову частину `st` у дане з типом `double`. Додатково повертає через `end` адресу першого символа, записаного за числом.
`double strtof (st, end);`*	Аналог `strtod()`, але повертає значення з типом `float`.
`double strtold (st, end);`*	Аналог `strtod()`, але повертає значення, що має тип `long double`.

* реалізовано в бібліотеках, що підтримують стандарт С-99

Наступний приклад демонструє застосування функції `strtol()` для виділення зі заданого символьного рядка всіх цілих чисел.

```
/*********************************************/
/*  Виділення всіх чисел із символьного рядка  */
/*********************************************/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main (void)
{
    char  str[] = "A=17280, B=-5120. Summa (12160) >0";      /* рядок */
    char  *p = str;
  printf("\nЗнайдено числа:  ");
  do {
    if ( isdigit(*p) || *p == '-' && isdigit(*(p+1)) ) /* знайдено число  */
       printf("%ld  ", strtol(p, &p, 0));          /* виведення числа */
    else                                      /* і перехід на його кінець */
       p++;
  } while ( *p != '\0' );
  return 0;
}
```

Результат виконання:

```
Знайдено числа:  17280  -5120  12160  0
```

Звернемо увагу на параметри у звертанні до функції `strtol()`:

```
strtol (p, &p, 0)
```

Перший параметр `p` є вказівником на початок поточного числа в рядку `str`. Другий параметр `&p` передає у функцію адресу вказівника `p`, за якою функція запише адресу першого нецифрового символа за знайденим числом. Таким чином, кожен виклик `strtol()` не тільки повертає значення знайденого довгого цілого числа, але й встановлює вказівник `p` на символ, з якого треба починати пошук наступного числа.

Перетворення "число ⇒ рядок". Бібліотека системи програмування Borland C додатково містить ряд функцій зворотних перетворень "число ⇒ символьний рядок" (див. табл. Д2.4 у Додатку 2), серед яких:

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

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

тільки тоді, коли значення `base` дорівнює `10`. У разі інших основ числення двійкові коди від'ємних чисел розглядаються і перетворюються як беззнакові.

Якщо потрібно записати в символьний рядок дійсне число або послідовність із декількох чисел, то зручно скористатись стандартною функцією виведення `sprintf()`. Ця функція є повним аналогом функції `printf()`, але виконує форматний запис даних у стрінг, адресу якого задає перший параметр функції. Ось приклад:

```
double  zm = 14.7063;
char  stzm[30];
sprintf (stzm, "%1.2f", zm);
```

Після виконання `sprintf()` у змінну `stzm` буде занесено символьний рядок `"14.71"` (значення `zm` за специфікацією `%1.2f`) з кінцевим `'\0'`.

Перетворення дійсних чисел у символьні рядки виконують три функції з бібліотеки Borland C: `ecvt()`, `fcvt()` та `gcvt()`, описи яких подано в табл. Д2.4 Додатка 2.

9.5. Масиви символьних рядків і масиви вказівників

Практичні задачі часто вимагають опрацювання груп символьних рядків. У таких випадках створюють або масиви символьних рядків, або масиви вказівників на перші символи рядків, а самі рядки розташовують в оперативній пам'яті окремо. Розглянемо обидва підходи.

9.5.1. Масиви символьних рядків

Поширеним видом багатовимірних масивів є масиви символьних рядків, тобто масиви, кожен елемент яких – окремий символьний рядок. Наприклад, масив найменувань міст, оголошений і проініціалізований наступним чином:

```
char cities[6][20] = {"Львів", "Хмельницький", "Полтава",
                      "Рівне", "Івано-Франківськ", "Київ"};
```

є двовимірним масивом з шести рядків, кожен з яких заповнений найменуванням відповідного міста (рис. 9.2).

Правила звертання до елементів масиву символьних рядків такі ж, як і для всіх інших багатовимірних масивів. Зокрема, перший рядок масиву (`"Львів"`) виділяють вирази `*cities` та `cities[0]`.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
cities:	Л	ь	в	і	в	\0														
	Х	м	е	л	ь	н	и	ц	ь	к	и	й	\0							
	П	о	л	т	а	в	а	\0												
	Р	і	в	н	е	\0														
	І	в	а	н	о	-	Ф	р	а	н	к	і	в	с	ь	к	\0			
	К	и	ї	в	\0															

Рис. 9.2. Заповнення масиву символьних рядків `cities[6][20]`

Наступні вирази виділяють першу літеру найменування останнього міста – літеру 'K':

```
cities[5][0] = *cities[5] = **(cities+5)    /* перша літера рядка 5 */
```

Наведемо програму, яка формує символьний рядок з двійково-десятковим кодом заданого довгого цілого беззнакового числа (у двійково-десятковому коді кожна десяткова цифра числа позначається відповідною двійковою тетрадою). Для формування коду у програмі використано масив символьних рядків codetetr, елементи якого зберігають двійкові тетради десятких цифр (від 0 до 9). Спочатку виділяються цифри числа і заносяться в масив narr. Далі за допомогою бібліотечної функції strcat() код двійкової тетради кожної цифри числа, починаючи від найстаршої, долучається до символьного рядка результату code2_10 (початково він порожній "").

```
/*********************************************/
/*  Формування двійково-десяткового коду числа  */
/*********************************************/
#include <stdio.h>
#include <string.h>                          /* для функції strcat() */
int  main (void)
{
     unsigned long  numb, ncopy;          /* задане число і його копія */
     char  code2_10[10*4+1] = "";        /* рядок для запису коду числа */
     unsigned  narr[10], *p;                      /* масив цифр числа */
     char  codetetr[][5] = {"0000",          /* масив 2-10-х кодів цифр */
                            "0001", "0010", "0011", "0100", "0101",
                            "0110", "0111", "1000", "1001"};
   printf ("\nЧисло - ");
   scanf ("%lu", &numb);
   ncopy = numb;           /* робоча копія числа, щоб зберегти оригінал */
   p = narr;                              /* вказівник на цифри числа */
   while (ncopy>0) {                     /* цикл виділення цифр числа */
      *p = ncopy%10;                 /* запис наймолодшої цифри в narr */
      ncopy/=10;                        /* відкидання виділеної цифри */
      p++;
   }        /* в масиві narr цифри числа записані у зворотному порядку */
   for (--p;  p>=narr;  p--)           /* цикл формування рядка коду */
      strcat (code2_10, codetetr[*p]);  /* долучення до рядка коду числа
                                 тетради з 2-10-м кодом відповідної цифри */
   printf ("2-10-й код числа %ld - %s\n", numb, code2_10);
   return 0;
}
```

Приклад виконання:

```
Число - 79065
2-10-й код числа 79065 - 01111001000001100101
```

Оскільки всі рядки масиву codetetr мають однакову довжину, то ділянка оперативної пам'яті, виділена для цього масиву, буде зайнята повністю. Проте так буває

не завжди. У випадках, коли символьні рядки мають різну довжину, розмірність рядка масиву має дорівнювати кількості символів найдовшого можливого стрінга плюс один байт для нуль-символа. Відповідно в рядках, де зберігаються короткі стрінги, більшість виділених байтів залишиться незаповненою, що добре видно з рис. 9.2.

9.5.2. Масиви вказівників на символи рядків

Створення масиву вказівників на перші символи набору рядків, кожен з яких зберігається в оперативній пам'яті окремо, – це альтернативний підхід, що дає змогу усунути недолік виділення зайвого місця, властивий масивам символьних рядків.

Розглянемо програму, яка здійснює випадковий вибір одного міста зі заданого списку міст (імітація жеребкування). В програмі створено масив вказівників `cparr`, елементи якого мають тип `char *`. Масив проініціалізовано стрінговими константами з найменуваннями міст. Кожному рядку, тобто найменуванню міста, компілятор виділяє в пам'яті ділянку, обсяг якої дорівнює довжині рядка плюс нуль-символ (рис. 9.3). Адреса розміщення рядка заноситься у відповідний елемент масиву `cparr`.

```
/*************************************/
/*   Вибір міста шляхом жеребкування  */
/*************************************/

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

int  main (void)
{
     char *  cparr[] = {"Львів", "Хмельницький",          /* список міст */
                        "Полтава", "Рівне", "Івано-Франківськ", "Київ"};
     int  rc;                                    /* номер вибраного міста */
   randomize();                                            /* випадковий  */
   rc = random(sizeof cparr / sizeof(char *));             /* вибір міста */
   printf ("\nВибране місто - %s.\n", cparr[rc]);
   return 0;
}
```

Приклад виконання:

```
Вибране місто - Полтава.
```

Рис. 9.3. Застосування масиву вказівників на символьні рядки

Щоб забезпечити можливість зміни складу і кількості міст, у програмі в оголошенні масиву `cparr` не вказано розмірність цього масиву. Тому задана кількість міст обчислюється за допомогою виразу

```
sizeof cparr / sizeof(char *)
```

Значення цього виразу використано як параметр функції `random()`, яка генерує випадкове число, що визначає номер вибраного міста.

Перевага використання масивів вказівників, які зберігають адреси початків рядків, замість масивів символьних рядків особливо відчутна, коли в програмі треба переставляти окремі рядки місцями, наприклад, потрібно впорядкувати набір рядків. У цих випадках самі символьні рядки не переписують (це важливо, адже їх розмір може бути достатньо великим), а тільки міняють місцями вказівники на початки рядків. Рис. 9.4 ілюструє результат перестановки вказівників у масиві `cparr`, що відповідає впорядкуванню найменувань міст за абетковим порядком.

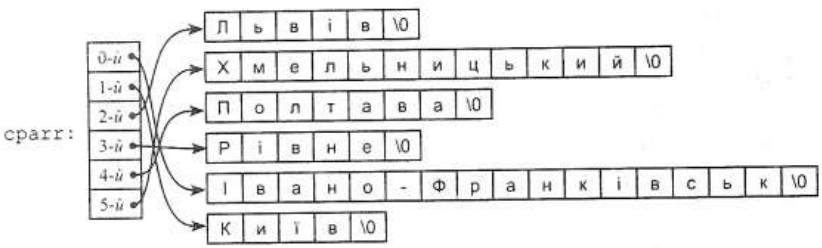


Рис. 9.4. Результат перестановки вказівників для впорядкування символьних рядків

9.5.3. Збереження символьних рядків у динамічній пам'яті

Масиви вказівників широко використовують для роботи зі символьними рядками та іншими даними, розташованими в динамічній пам'яті. Як вже зазначалось, серед функцій бібліотеки Borland C, прототипи яких записані в заголовному файлі `<string.h>`, є спеціальна функція `strdup()`, призначена для запису в динамічну пам'ять заданого рядка символів, включаючи нуль-символ. Функція виділяє в динамічній пам'яті ділянку потрібного обсягу, записує туди копію рядка і повертає адресу першого символа перенесеного символьного рядка.

Далі подано програму, яка вводить послідовність символьних рядків (речень) і відразу записує кожен рядок у динамічну пам'ять. Адреси рядків заносяться у масив вказівників `mp`. Ознакою завершення введення слугує порожній рядок. Останнє слово останнього введеного речення є ключовим. Програма виділяє його і друкує всі речення, які містять дане слово. Щоб зосередити увагу на роботі з масивом вказівників і спростити програму, передбачено ряд обмежень на вхідні дані: останнє речення складається з декількох слів; після ключового слова не повинно бути інших символів, зокрема розділових знаків; ключове слово не повинно зустрічатися як частина іншого слова; літери ключового слова в усіх введених рядках повинні мати однаковий регістр.

```
/*************************************************************/
/*  Запис речень у динамічну пам'ять, пошук ключового слова  */
/*************************************************************/
#include <stdio.h>
#include <string.h>
#define NMAX 50                  /* максимальна кількість речень */
#define STLEN 130                /* максимальна довжина речення */
#define KWLEN 20                 /* максимальна довжина ключового слова */
#define STEND ""                 /* кінець введення - порожній рядок */
int main (void)
{
    char *mp[NMAX],              /* масив вказівників на речення в ДП */
         sent[STLEN],            /* масив для введення речень */
         keywrd[KWLEN];          /* масив для ключового слова */
    char **pp, *pkw, *plast;     /* робочі вказівники */
    int k=0;                     /* кількість введених речень */
/*  цикл введення речень і запису їх у динамічну пам'ять  */
  puts("\n\tРечення (кінець - порожній рядок): ");
  do {
     gets(sent);
     if (strcmp(sent, STEND) == 0)          /* введено порожній рядок */
        break;
     mp[k] = strdup(sent);              /* запис рядка в динамічну пам'ять */
  } while (++k < NMAX);
  plast = mp[k-1];                        /* вказівник на останній рядок */
  pkw = strrchr(plast, ' ')+1;               /* пошук останнього слова */
  strcpy(keywrd, pkw);                 /* запис ключового слова в keywrd */
  printf("\n\tРечення з ключовим словом %s:\n", keywrd);
  for (pp=mp; *pp!=plast; pp++)   /* цикл перевірки введених речень */
     if (strstr(*pp, keywrd)!= NULL)     /* в реченні є ключове слово */
        puts(*pp);                /* pp - вказівник на елементи масиву mp */
  return 0;
}
```

Приклад виконання:

```
    Речення (кінець - порожній рядок):
Перші ЕОМ з'явились в інституті наприкінці 60-х років.
Усі вони були ламповими.
ЕОМ використовували тільки для спеціальних розрахунків.
Програмування здійснювалось у машинних ходах.
Ключове слово - ЕОМ
 < порожній рядок, що є ознакою кінця введення >

    Речення з ключовим словом ЕОМ:
Перші ЕОМ з'явилися в інституті наприкінці 60-х років.
ЕОМ використовували тільки для спеціальних розрахунків.
```

Для опрацювання введених символьних рядків у попередній програмі використано декілька бібліотечних функцій із заголовного файла <string.h>. Зокрема, звертання до функції strcmp():

```
if (strcmp(sent, STEND) == 0)  break;
```

перевіряє, чи введений символьний рядок sent збігається з рядком ознаки завершення циклу введення STEND. Запис введених речень у динамічну пам'ять виконує функція strdup():

```
mp[k] = strdup(sent);
```

Адреса k-го речення заноситься у відповідний за номером елемент масиву вказівників mp. Нагадаємо, що strdup() не належить до стандартних функцій мови С; інші функції, призначені для роботи з динамічною пам'яттю, розглядатимуться в розділах 13 і 14.

Оператор

```
pkw = strrchr(plast, ' ')+1;
```

звертається до функції strrchr() для пошуку останнього символа пробілу в останньому введеному рядку, адресу якого записано у вказівник plast. За цим пробілом починається ключове слово. Виклик функції strcpy():

```
strcpy(keywrd, pkw);
```

призначений для копіювання ключового слова у масив keywrd.

Щоб перевірити, чи входить ключове слово keywrd у введені речення, використано функцію strstr():

```
if (strstr(*pp, keywrd))   puts(*pp);
```

Змінну pp оголошено як вказівник на вказівник на дані з типом char:

```
char **pp;
```

Після присвоєння pp=mp значенням виразу *pp є значення елемента масиву mp, на який вказує pp, тобто адреса відповідного символьного рядка в динамічній пам'яті. Операція pp++ просуває вказівник pp на наступний елемент масиву mp. Цикл завершується, коли pp досягає елемента, який вказує на останній символьний рядок, значення цього елемента збігається зі значенням вказівника plast.

Запитання та завдання для самоконтролю

1. Який вид даних називається символьним рядком? Яке призначення символа '\0'?
2. У програмі необхідно опрацювати символьний рядок, що вводиться з клавіатури. Яке з наступних оголошень змінної instr, у якій повинен зберігатись цей рядок, є правильним?

   1) `char *instr;`
   2) `char instr[];`
   3) `char instr[200];`
   4) `char *instr[200];`

   У чому помилковість інших оголошень?

3. У програмі оголошено:

```
char *ps="Перевіряємо свої знання";
```

Яке значення матиме кожен із наступних виразів?

1) `*ps`
2) `ps[11]`
3) `*(ps+20)=='\0'`
4) `*ps+2<'Я'`
5) `strlen(ps)`
6) `strlen(ps+17)`

4. Які значення повертають стандартні бібліотечні функції `getchar()` і `gets()`? В яких випадках краще використовувати `gets()`, а в яких доцільніше вводити символьні рядки через `scanf()`?
5. Для чого призначені бібліотечні функції, оголошені у `<ctype.h>`?
6. Який заголовний файл необхідно підключити до програми для роботи з бібліотечними функціями опрацювання символьних рядків? Назвіть декілька з цих функцій та дії, які вони виконують?
7. У чому відмінність двох наступних оголошень?

```
char ms1[10][150];
char *ms2[10];
```

Коли доцільніше формувати масиви символьних рядків, а коли – масиви вказівників на перші символи рядків?

8. Чому групи символьних рядків доцільно зберігати в динамічній пам'яті?

Запрограмуйте подані нижче задачі, використовуючи різні форми звертання до символів рядка і символьних рядків у цілому

9. Визначити, скільки разів у введеному з клавіатури символьному рядку повторюється заданий символ.
10. Задано символьний рядок, який є двійковим кодом цілого беззнакового числа (складається із символів 1 та 0). Обчислити і вивести на екран десяткове значення цього числа.
11. Задано речення. Виділити з нього і записати в окремий масив слово, порядковий номер якого задає користувач. Надрукувати це слово.
12. З клавіатури ввести речення. Сформувати нове речення зі зворотним порядком слів.
13. Ввести і записати в динамічну пам'ять послідовність речень українською мовою, сформувавши масив вказівників на перші символи рядків. Визначити, чи є літери, які використовувались у всіх введених реченнях. Надрукувати ці літери.
14. Введений з клавіатури символьний рядок є арифметичним виразом, що має таку форму:

число_1 ⊕ *число_2*

тут *число_1* і *число_2* – два довільних дійсних числа, ⊕ – один із знаків арифметичних операцій: +, -, *, / (перед знаком і за ним може бути записаний символ пробілу). Виділити із введеного рядка операнди й знак операції та обчислити значення цього виразу.

Розділ 10

СТРУКТУРИ ТА ОБ'ЄДНАННЯ

У цьому розділі:

- Поняття структури, шаблони структур
- Структурні змінні: оголошення, ініціалізація, розміри; операція присвоєння для структур
- Звертання до елементів структур через операцію "крапка" та через вказівник на структуру й операцію"стрілка"
- Вкладення структур, масиви структур, вказівники на структури та вказівники як поля структур
- Декларація перейменування типів `typedef`
- Об'єднання: означення, розташування в оперативній пам'яті, розмір, оголошення шаблонів і змінних, ініціалізація
- Операції над об'єднаннями, звертання до полів об'єднання, використання об'єднань для виділення окремих частин даних
- Поля бітів: синтаксис, призначення, приклад застосування

Агреговані типи даних, що розглядаються у даному розділі, а саме: структури, об'єднання, поля бітів, а також декларація `typedef` – дають змогу створювати т. зв. *користувацькі* типи даних, імена й наповнення яких задаються розробником програми. Такі типи істотно розширюють можливості вибору форм збереження даних і помітно спрощують процес програмування.

10.1. Структури

Структури – це особливий комбінований тип даних, який об'єднує у спільне ціле набір логічно пов'язаних між собою різнотипних компонентів. Складові частини структури називають *полями* або *елементами*. Кожне поле структури має свій тип і своє ім'я.

10.1.1. Оголошення та ініціалізація структур

Зі структурами в С-програмах пов'язані два поняття:

- *шаблон структури* – визначає конкретну структуру та задає склад і послідовність запису її полів;

- *структурні змінні* – дані, сформовані за вказаним шаблоном, для яких в оперативній пам'яті виділено ділянку відповідного обсягу.

Шаблони структур. Шаблон структури оголошується наступною синтаксичною конструкцією:

```
struct тег_структури {
   тип_поля1 ім'я_поля1;
   тип_поля2 ім'я_поля2;
   . . .
   тип_поляk ім'я_поляk;
};
```

тут `struct` – службове (ключове) слово, що специфікує структуру; *тег_структури* – ім'я, яким позначатимуть у програмі структури даної форми. Список полів структури охоплюється фігурними дужками {}, після правої дужки } записується знак ; (за умови, що після шаблону структури не оголошено структурних змінних). Кожне поле структури описується як змінна – типом та іменем.

Доповнимо сказане. *Тег* (ярлик, ім'я) надається шаблону структури програмістом, щоб надалі в програмі можна було ідентифікувати структури вказаної форми і складу. Теги записуються як звичайні ідентифікатори мови С. Якщо в програмі використовується тільки одна структура, то її шаблон можна оголосити без тега:

```
struct {
   список полів структури
};
```

Такий шаблон називають безіменним. Можна залишити безіменним один з усіх оголошених у програмі шаблонів структур.

Поля структури описуються послідовно. В оперативній пам'яті вони будуть записані саме в тому порядку, в якому були задані в шаблоні даної структури. Тип поля може бути довільним простим або складеним типом (більш детально про це мова йтиме далі). Ім'я поля є ідентифікатором змінної відповідного типу. В межах одного шаблону імена всіх полів повинні бути різними, але вони можуть збігатися з іменами полів інших структур чи з іменами змінних програми.

Для прикладу оголосимо шаблон структури, в яку можна буде записувати інформацію про студентів – учасників конкурсу:

```
struct student {
   char name[40];
   char grup[8];
   long int number;
   double rating;
};
```

Тегом даної структури є ім'я `student`. Структура складається з чотирьох полів. Перші два поля `name` та `grup` є символьними рядками, в яких буде зберігатись інформація

про прізвище й ім'я студента та його академгрупу. Поле `number` цілочислове, воно призначене для запису номера студентського квитка. Останнє поле `rating` має дійсний тип, у це поле буде заноситись середній бал успішності студента.

За своїм змістом шаблон структури – це опис нового типу, створеного програмістом. Для шаблону не виділяється місце в оперативній пам'яті, над ним не можна виконувати ніяких операцій. Шаблон слугує основою для оголошення змінних зі створеним структурним типом.

Структурні змінні. Застосовують дві форми оголошення структурних змінних:

- через посилання на попередньо оголошений шаблон;
- одночасно з оголошенням шаблону структури.

Перша форма передбачає, що шаблон структури вже оголошено. Тоді його тег спільно зі словом `struct` можна використовувати для створення змінних відповідного структурного типу. Зокрема, використовуючи оголошений шаблон структури `student`, можна оголосити відповідні структурні змінні:

```
struct student  stud1, stud2;
struct student  sarr[50];
```

У результаті даних оголошень в оперативній пам'яті буде виділено місце для двох змінних `stud1` і `stud2` та для масиву `sarr`, що складається з 50-ти елементів. Обидві змінні та всі елементи масиву `sarr` будуть мати однаковий розмір і склад полів, встановлений в оголошенні шаблону структури `student`.

Мова С дозволяє оголошувати структурні змінні одночасно з оголошенням шаблону структури. Наприклад, структурні змінні можна оголосити таким чином:

```
struct point {
   int x;
   int y;
} upleft, downright, polygon[8];
```

У наведеному прикладі, як і в попередньому, оголошено дві змінні та масив, але їх оголошення виконано разом із записом шаблону структури, призначеної для збереження координат точки площини. Кожна зі змінних `upleft` і `downright` (це можуть бути координати лівого верхнього і правого нижнього кутів прямокутника) буде складатись із двох цілочислових полів `x` та `y`. Масив `polygon` сформовано з восьми елементів, що мають тип `struct point` (елементи масиву можуть бути координатами вершин багатокутника).

Ініціалізація структур. В оголошеннях структурні змінні можна ініціалізувати, тобто присвоювати їх елементам початкові значення. Як і у випадку масиву, список значень, якими ініціалізується структура, вказується у фігурних дужках.

Приклад ініціалізації структурної змінної:

```
struct student  best_stud = { "Гончаренко Андрій", "КН-41", 2160473,
                              96.58 };
```

Значення, якими ініціалізується структура, записують строго в тій самій послідовності, в якій вказані поля у шаблоні структури, а типи констант-ініціалізаторів повинні бути сумісним з типами відповідних полів. Можна виконувати часткову ініціалізацію – задавати значення лише декількох початкових елементів структури. Стандарт С-99 надав додаткову можливість вибіркової ініціалізації полів структури (див. Додаток 3).

Якщо ініціалізується масив структур, то послідовно записуються значення, якими ініціалізується кожна зі структур, що входять до складу масиву, наприклад:

```
struct point  outline[] = {{40, 45}, {86, 72}, {64, 23}};
```

Кількість елементів масиву `outline` буде визначатись кількістю ініціалізаторів-структур, для даного прикладу вона становитиме 3. Якщо ініціалізуються всі поля структур, то внутрішні фігурні дужки можна опустити. Зокрема, ініціалізація масиву `outline` може бути такою:

```
struct point  outline[] = {40, 45, 86, 72, 64, 23};
```

Очевидно, що ця форма ініціалізації менш наочна, ніж перша, і це може стати причиною помилок у записі значень, якими ініціалізуються структури.

10.1.2. Розмір структури. Операція присвоєння для структур

Розміри структурних змінних. Для кожної структурної змінної в оперативній пам'яті виділяється неперервна ділянка, обсяг якої дорівнює або більший за сумарний розмір усіх полів даної структури. Приміром, розмір змінної з типом `struct point` буде не меншим за 4 байти, а розмір змінної з типом `struct student` – не меншим за 60 байтів. Реальний обсяг ділянки, яку займає структурна змінна, залежить від прийнятого в даній системі способу вирівнювання даних. Залежно від апаратно-програмних установок вирівнювання може виконуватись на межу слова, подвійного слова або не виконуватись взагалі. У разі відсутності вирівнювання поля структури записуються підряд, інакше між полями можуть залишатись незайняті байти. В інтегрованому середовищі Borland C в розділі меню Options/Compile/Code Generation є опція Alignment, яка дає змогу зняти вирівнювання або встановити його на межу слова – тоді кожне нове поле буде починатись з парної адреси.

У програмі розмір структури треба визначати операцією `sizeof`. Нагадаємо, що конструкція `sizeof(`*тип*`)` потребує дужок, а для виразу `sizeof` *змінна* у мові С дужки не обов'язкові.

Наприклад, значенням виразу `sizeof(struct student)` буде розмір кожної змінної, що має тип `struct student`. Такий самий результат дадуть вирази: `sizeof best_stud`, `sizeof stud1`, а також `sizeof sarr[0]`.

Присвоєння структур. Ще однією важливою властивістю структурних змінних є те, що вони розглядаються як звичайні змінні, а не як вказівники. Тому, якщо дві структурні змінні мають спільний шаблон, а отже займають однакові за обсягом ділянки в

оперативній пам'яті, то можна присвоїти значення однієї структури іншій. Реалізація присвоєння полягає в повному копіюванні ділянки структури, вказаної справа від знака присвоєння, в ділянку структурної змінної, вказаної зліва. Оператор присвоєння

```
stud1 = best_stud;
```

перепише вміст усіх полів змінної `best_stud` у структуру `stud1`, а оператор

```
polygon[5] = *outline;
```

скопіює в шостий елемент масиву `polygon` вміст першої структури масиву `outline`. Ще раз звернемо увагу на те, що в операції присвоєння обидві структури повинні мати однаковий тип, тобто спільний шаблон.

Інші операції (крім операції визначення адреси `&`, про яку мова йтиме далі) над цілими структурами не допускаються. Зокрема, не можна порівнювати структури. Всі дії щодо опрацювання структур виконуються над їх елементами.

10.1.3. Вкладені структури, масиви структур, вказівники на структури

Вкладення структур. Вже зазначалося, що поля структур можуть мати довільний тип, як простий, так і складений, зокрема поле може бути масивом, символьним рядком (як у структурі `struct student`) чи вкладеною структурою. Для внутрішніх (вкладених) структур діють такі ж правила оголошення, як і для зовнішніх: можна окремо оголошувати шаблон структури або задавати його всередині зовнішньої структури разом з оголошенням відповідних змінних.

Наведемо приклад, що ілюструє вкладення структур. Нехай у програмі крім структури `struct point`, опис якої було наведено раніше, оголошено також структуру зі шаблоном `RGB`:

```
struct RGB  {
   int red, green, blue;
};
```

поля якої задають інтенсивність червоної, зеленої і синьої складових кольору зображення. Ще один шаблон:

```
struct circle  {
   struct point  center;
   unsigned int  radius;
   struct RGB  color;
};
```

визначає структуру, яка описує коло. Поля `center` і `color` цієї структури є вкладеними структурами, тобто їх розмір і склад задаються попередньо оголошеними шаблонами `struct point` і `struct RGB`. Ініціалізацію відповідної структурної змінної можна виконати так:

```
struct circle  bigc = {{10,-40}, 180, {40, 18, 40}};
```

Першу пару значень, яка задає координати точки центра кола змінної `bigc`, та останню трійку, що задає інтенсивності RGB-складових кольору, записано в `{}`, аби підкреслити, що це вкладені структури, хоча в разі повної ініціалізації полів структурної змінної внутрішні дужки не є обов'язковими.

Масиви структур. З наборів однотипних структур можна формувати масиви. Повторимо один із наведених раніше прикладів оголошення масиву структур:

```
struct point  outline[] = {{40, 45}, {86, 72}, {64, 23}};
```

Оскільки в оголошенні не вказано явно розмірність масиву `outline`, то вона встановлюється компілятором за кількістю проініціалізованих структур. Щоб отримати значення кількості елементів масиву програмно, треба скористатись виразом з операціями `sizeof`:

```
sizeof (outline) / sizeof (struct point)
```

або

```
sizeof (outline) / sizeof (*outline)
```

Масиви структур повністю зберігають всі базові властивості масивів мови С. До елементів такого масиву можна звертатись через індекси або через вказівники. Зокрема, вираз `*outline` рівнозначний виразу `outline[0]` – його значенням є перша з усіх структур масиву `outline`. Відповідно вираз `outline[3]` або `*(outline+3)` визначає структуру, індекс якої в масиві дорівнює 3.

Вказівники на структури. Оскільки структурні змінні не є вказівниками, а розглядаються як звичайні змінні програми, то до них можна застосовувати операцію визначення адреси `&`. Результатом операції буде адреса першого байта ділянки, яку займає в оперативній пам'яті структура-операнд. Значення адреси структури можна присвоїти вказівнику, базовий тип якого збігається з типом структурної змінної.

Якщо в програмі оголошено вказівник

```
struct point  *pstrp;
```

то йому можна присвоїти адресу кожної зі структурних змінних, що мають шаблон `struct point`. Наприклад, можна виконати таке присвоєння:

```
pstrp = &upleft;
```

У разі виконання наступного оператора:

```
pstrp = outline;          /* або   pstrp=&outline[0]; */
```

значення вказівника `pstrp` дорівнюватиме адресі першої структури з масиву `outline`. А в разі збільшення вказівника

```
pstrp++;
```

він пересунеться на наступний елемент масиву, тобто значенням `pstrp` стане адреса першого байта наступної структури з масиву `outline`.

10.1.4. Звертання до елементів структур

Мова С має дві операції, призначені для звертання до елементів (полів) структури:

- операцію "крапка", яку позначають знаком . і застосовують у звертаннях до поля структури, заданої через змінну;
- операцію "стрілка", яку позначають парою знаків -> (мінус і більше) та використовують для звертання до поля структури через вказівник на цю структуру.

Розглянемо детальніше обидві форми звертання.

Операція "крапка". Виділення поля структури, заданої через змінну, виконує така синтаксична конструкція (вираз):

ім'я_структурної_змінної.*ім'я_поля*

Цей вираз називають *уточненим іменем* елемента структури. Його значенням є значення змінної, яка формує відповідне поле структури. Наприклад, звертання до полів структури `best_stud`, оголошення та ініціалізацію якої ми наводили раніше, дадуть такі результати:

`best_stud.name` – рядок із 40-а символів, у який занесено стрінг `"Гончаренко Андрій"`;
`best_stud.number` – число з типом `long int`, що має значення `2160473`;
`best_stud.rating` – дійсне число з типом `double`, що дорівнює `96.58`.

Поля структури повністю зберігають властивості даних свого типу, над ними можна виконувати всі операції, що допускаються для цього типу. Зокрема, щоб змінити значення елементів структури `best_stud`, треба враховувати особливості типу кожного з її полів:

```
best_stud.rating = newrating;                /* зміна поля rating */
```

– тут використовуємо звичайне присвоєння, оскільки поле `rating` є числовим;

```
strcpy(best_stud.grup, newgrupname);         /* зміна поля grup */
```

– а тут застосовуємо функцію копіювання стрінгів, оскільки змінюємо значення цілого символьного рядка.

Хоча за допомогою операції присвоєння можна повністю переписати вміст однієї структури в іншу (включаючи всі поля-масиви і поля-рядки), окремо до полів-масивів і полів-рядків застосовувати присвоєння не можна, бо їх уточнені імена залишаються константними вказівниками на перші елементи цих масивів.

Обидві операції виділення елементів структури: . та -> мають однаковий найвищий рівень старшинства та асоціативність зліва направо, що є визначальним для формування виразів, у які входять елементи структур. Значенням виразу

```
*best_stud.name                              /* перша літера name */
```

є перша літера рядка, в який записано прізвище студента. Операція виділення поля `name`, позначена крапкою, має вище старшинство і виконується першою, а операція розадресації * застосовується до уточненого імені рядка і повертає його перший символ

(літеру 'Г'). Відповідно значенням виразу

```
best_stud.name[11]          /* дванадцята літера name */
```

є літера 'A' – символ з індексом 11 із рядка name. В даному випадку операції . та []: виділення поля структури та виділення елемента масиву – мають однакове старшинство, тому виконуються згідно з правилами асоціативності зліва направо, тобто спочатку виділяється рядок, а потім відбувається звертання до символа, що має індекс 11. Таке саме значення матиме вираз

```
*(best_stud.name+11)        /* дванадцята літера name */
```

який є адресною формою звертання до елемента масиву.

Відзначимо принципову відмінність виразів

```
stud1.grup[3]
```

та

```
sarr[3].grup
```

Обидва вирази звертаються до поля grup структур, що мають шаблон student. Але значенням першого виразу є четверта за порядком літера найменування групи з даних структури stud1, а значенням другого виразу є весь символьний рядок grup (точніше, вказівник на його перший символ), тобто другий вираз повертає найменування групи четвертого студента з масиву sarr.

Якщо поле структури є вкладеною структурою, то для звертання до полів внутрішньої структури операцію "крапка" застосовують кількаразово, відповідно до глибини вкладення структур. Наприклад, щоб виділити координати центра кола, параметри якого задаються структурою bigc, описаною в попередньому параграфі, треба використати конструкції:

bigc.center.x – горизонтальна координата центра кола, її значення після ініціалізації bigc дорівнює 10;

bigc.center.y – вертикальна координата центра кола, що має значення 40.

Оператор

```
bigc.color.green *= 2;
```

який звертається до поля green внутрішньої структури color, збільшить удвічі інтенсивність зеленої складової кольору кола.

Коли звертаються до полів структури, яка є елементом масиву структур (таким, наприклад, як описані раніше outline чи sarr), то для доступу до елемента масиву застосовують як індексну, так і вказівникову форму звертання. Обидва вирази:

```
sarr[k].number
```

та

```
(*(sarr+k)).number
```

рівнозначні, хоча очевидно, що другий вираз малонаочний і важчий для сприйняття. Зовнішні дужки в записі (*(sarr+k)) необхідні з огляду на старшинство операції "крапка" щодо операції розадресації *.

Операція "стрілка". Операція -> спрощує звертання до полів структур, які виконуються через адреси цих структур або вказівники на структури. Синтаксис її такий:

вказівник_на_структуру -> ім'я_поля

Вираз `sarr+k` з попереднього прикладу є адресою k-ї структури з масиву `sarr`, тому для звертання до поля `number` даної структури можна використати операцію "стрілка":

```
(sarr+k) -> number
```

Усі записані далі вирази матимуть однакове значення – першу літеру прізвища першого студента з масиву `sarr`:

```
sarr[0].name[0]     *sarr[0].name          *(*sarr).name
sarr -> name[0]     *(&sarr[0]) -> name    *sarr -> name
```

Вказівникова форма звертання особливо ефективна, коли для роботи з масивом структур використовують зовнішні вказівники, базовий тип яких збігається з типом структур – елементів масиву.

Щоб практично проілюструвати все сказане, розглянемо приклад програми, яка опрацьовує масив структур із шаблоном `struct student`, використовуючи різні форми звертання до полів структур. У програмі спочатку здійснюється введення з клавіатури і запис у масив `starr` даних групи студентів. Потім введені дані впорядковуються за спаданням рейтингу успішності студентів. Результат упорядкування виводиться на екран у формі таблиці. Додаткові пояснення подано після програми.

```
/*******************************************/
/*  Формування і сортування масиву структур  */
/*******************************************/
#include <stdio.h>
#define  KST  25                    /* максимальна кількість студентів */
int main(void)
{
  struct student {                                /* шаблон структури */
     char name [35];
     char grup[10];
     long number;
     double rating;
  } starr[KST], stud, *pst, *pst1, *pst2;        /* структурні змінні */
  int k, kst;
  double inrating;
   printf("Кількість студентів - ");
   scanf("%d",&kst);    getchar();
   puts("\tДані: ");
   for(k=0;  k<kst;  k++){                     /* цикл введення даних */
      printf("\n%2d.  Прізвище, ім\'я: ", k+1);
      gets(starr[k].name);
```

```
      printf("группа: ");
      gets((starr+k)-> grup);
      printf("номер студентського квитка: ");
      scanf("%ld",&starr[k].number);
      printf("середній бал: ");
      scanf("%lf",&inrating);  getchar();
      (starr+k) -> rating = inrating;
   }
   for(k = 0;  k < kst;  k++) {                          /* цикл сортування */
      pst1 = starr;     pst2 = starr+1;
      for( ;  pst2 < starr+kst-k;  pst1++, pst2++ )
         if (pst1 -> rating < pst2 -> rating) {  /* якщо сусідні струк- */
            stud=*pst1;                          /* тури не впорядковані, */
            *pst1 = *pst2;                       /* то міняємо їх місцями */
            *pst2=stud;
      }
   }
   puts("\n\t Список студентів: ");
   for(k = 0, pst = starr;  k < kst;  k++, pst++)
      printf("%2d.  %-30.27s%-8s%8ld%8.2lf\n", k+1, pst -> name,
             pst -> grup, pst -> number, pst -> rating);
   return 0;
}
```

Приклад виконання:

```
Кількість студентів - 10
        Дані:
 1. Прізвище, ім'я: Вовчак Ірина
группа: КН-46
номер студентського квитка: 2160561
середній бал: 94.72
  . . .          (дані наступних студентів)

        Список студентів:
 1.  Гончаренко Андрій              КН-41    2160473   96.58
 2.  Вовчак Ірина                   КН-46    2160561   94.72
  . . .          (дані інших студентів)
```

Прокоментуємо коротко наведену програму, звернувши увагу на кілька практичних моментів. У процесі введення даних з клавіатури двічі використано функцію `getchar()`, щоб вилучити з буфера введення символ нового рядка, який залишається там після зчитування числа функцією `scanf()`. Якщо не зробити цього, то наступне введення прізвища студента через функцію `gets()` сприйме цей символ як ознаку порожнього рядка. Для повного очищення буфера введення можна скористатись функцією `fflush()`:

```
fflush(stdin);
```

(детальніше про цю функцію можна дізнатись у розділі 15).

Нагадаємо, що в списку введення функції `scanf()` вказуються адреси змінних. Зокрема, вираз `&starr[k].number` задає адресу поля `number` у структурі даних k-го студента.

Для введення дійсного числа, що задає середній бал успішності студента, в програмі використано додаткову змінну `inrating`.

Використання додаткової змінної пов'язане з тим, що функція `scanf()` з бібліотеки Borland C містить помилку, яка за певних умов (зокрема у випадках, коли в списку введення задається адреса поля структури, яке має дійсний тип, а сама структура є елементом масиву структур) може викликати аварійне завершення роботи програми з виведенням повідомлення:

```
scanf: floating point formats not linked
```

Помилку введення дійсного поля елемента масиву структур можна обійти також, використавши для введення даних окрему структурну змінну (таку як змінна `stud` у нашій програмі), вміст якої треба потім переписати у відповідний елемент масиву:

```
starr[k] = stud;
```

Сортування масиву структур у програмі виконується за методом попарного порівняння елементів: порівнюються рейтингові бали студентів і, якщо вони не відповідають спадному порядку, то дві сусідні структури переставляються місцями.

Вказівники як поля структур. Мова C не допускає, щоб вкладена в структуру внутрішня структура мала той самий тип (шаблон), що й структура, елементом якої вона є. Водночас поле структури може бути вказівником з довільним базовим типом, у тому числі й вказівником на структуру. Тому дозволено, щоб поле структури було вказівником на структуру свого типу. Наприклад, у разі оголошення:

```
struct element {
   char info[300];
   struct element *next;
} first, last;
```

вказівник `next` може зберігати адресу кожної структурної змінної, що має шаблон `struct element`. Якщо виконати присвоєння

```
first.next = &last;
```

то значенням виразу:

```
first.next -> info
```

буде вказівник на перший символ рядка `info` в структурі, адресу якої зберігає вказівник `next`, тобто значення цього виразу дорівнює `&last.info[0]`.

Структури з полями, які вказують на інші структури такого самого типу, є базовими для формування динамічних списків і дерев (динамічні інформаційні структури розглянемо в розділі 13).

10.2. Перейменування типів

До складу конструктивних елементів мови С входить спеціальна декларація `typedef`, яка дає програмістові змогу надавати власні імена типам даних програми. Наприклад, за допомогою `typedef` можна перейменувати тип `unsigned int` на коротшу форму запису – `UINT`:

```
typedef  unsigned int  UINT;
```

Надалі найменування типу `UINT` можна використовувати в усій програмі там, де треба оголошувати змінні з типом `unsigned int`:

```
UINT  max, min, numb, k;
```

Загальна конструкція декларації `typedef` така:

`typedef` *тип користувацьке_ім'я_типу;*

Всередині `typedef` *користувацьке_ім'я_типу* записується в тому місці, де для звичайного оголошення вказується ім'я змінної. Якщо в програмі задекларовано:

```
typedef  char STRING[300];
```

то тип `STRING` буде масивом, що складається із 300 елементів-символів. Цей тип можна використовувати для оголошення відповідних символьних рядків, а також типів, похідних від них. Наприклад:

```
STRING st1, st2, stnew[25];
```

За цим оголошенням `st1` і `st2` будуть символьними рядками, кожен з яких може містити до 300 символів, а `stnew` – двовимірним масивом, що складається із 25-и символьних рядків з типом `STRING`.

Декларація `typedef` не створює нового типу, а тільки змінює ім'я існуючого стандартного чи користувацького типу або іменує оголошений у декларації тип.

Використання декларації `typedef` сприяє підвищенню наочності імен типів даних та скорочує їх запис. Це передусім стосується таких типів як структури, об'єднання і переліки, для яких `typedef` можна використовувати одночасно з оголошенням шаблону. Наприклад, перейменувавши шаблон структури, що описує хімічний елемент:

```
typedef  struct chemical_element {
     char name[16];
     int number;
     char abravir[4];
     double mass;
} ChemEl;
```

надалі зможемо використовувати ім'я структурного типу `ChemEl` у всіх наступних оголошеннях:

```
ChemEl elemset[100], *pelem;
```

Крім наочності та простоти іменування типів, `typedef` надає програмістові змогу створювати машинонезалежні мобільні типи даних. Насамперед це стосується тих типів, параметри яких залежать від апаратних особливостей комп'ютера. Якщо такому типові надати певне `typedef`-ім'я (прикладом може слугувати декларація типу `UINT`) і використати це ім'я всюди в тексті програми, то в новому середовищі достатньо лише внести зміни в рядок декларації `typedef`, щоб встановити потрібні параметри для перейменованого типу. Наприклад, щоб тип `UINT` був двобайтовим цілим типом на комп'ютерах, для яких `int` є чотирибайтовим, треба в його декларацію додати слово `short`:

```
typedef  unsigned short int  UINT;
```

10.3. Об'єднання

Об'єднання – це особливий тип даних, який дає змогу записувати в одну і ту ж встановлену ділянку оперативної пам'яті дані різних типів і розмірів. Таким чином створюється ділянка пам'яті спільного користування, до якої можна звертатись різними способами через єдину змінну, що має тип об'єднання.

Оголошення об'єднання подібне до оголошення структури: задається шаблон об'єднання та перелічуються відповідні змінні. Так само, як і для структур, шаблон об'єднання можна оголошувати окремо або одночасно з оголошенням змінних. У разі автономного оголошення шаблону об'єднання застосовують синтаксичну конструкцію, аналогічну до шаблону структури, але починають її ключовим словом `union`:

```
union тег_об'єднання  {
   тип_поля1  ім'я_поля1;
   тип_поля2  ім'я_поля2;
   . . .
   тип_поляk  ім'я_поляk;
};
```

Тег_об'єднання ідентифікує дане об'єднання, а список полів задає перелік даних, які можна заносити в це об'єднання.

Як і у випадку структур, змінні з типом об'єднання можна оголошувати спільно з оголошенням шаблону або пізніше, використовуючи тип `union` *тег_об'єднання*. Обсяг ділянки пам'яті, яка виділяється для кожної змінної, що має тип об'єднання, визначається розміром найдовшого поля даного об'єднання.

У декларації `typedef` можна сумістити оголошення шаблону об'єднання та його найменування. Наведемо приклад оголошення іменованого об'єднання:

```
typedef union demo_union {
     double x;
     char ch;
     int a;
}  UNDEM;
```

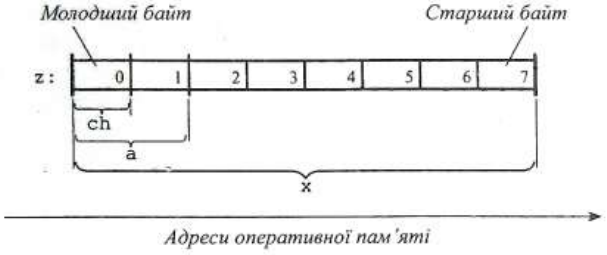


Рис. 10.1. Взаємне накладання полів об'єднання UNDEM

Якщо тепер оголосити змінну z, використовуючи задекларований тип UNDEM:

```
UNDEM z;
```

то така змінна буде об'єднанням із шаблоном demo_union, тобто всі три її поля: x, ch та a будуть користуватись спільною ділянкою оперативної пам'яті (рис. 10.1). Обсяг ділянки, яку займає змінна z, дорівнює розміру її найдовшого поля. У нашому випадку найдовше поле x, воно має тип double і займає 8 байтів.

Відзначимо, що всі три поля змінної z мають однакове нульове зміщення відносно адреси початку ділянки, яку займає z. Якщо молодший байт даних має адресу, меншу, ніж старший байт (такий порядок розташування даних властивий, зокрема IBM PC), то поля об'єднання накладаються одне на інше в області молодших адрес. Байт, який займає поле ch, збігається з молодшим байтом поля a та з наймолодшим байтом поля змінної x.

Оголошуючи змінну з типом об'єднання, можна відразу проініціалізувати цю змінну, тобто записати у її ділянку певне початкове значення. Звернемо увагу, що ініціалізувати об'єднання можна тільки значенням, сумісним з типом першого поля цього об'єднання (для змінної z це тип double). Як і у випадку масивів та структур, список значень, якими ініціалізується об'єднання (навіть якщо це буде тільки одне значення), записується у фігурних дужках. Наприклад, змінну z в оголошенні можна було б проініціалізувати так:

```
UNDEM z = { -436.05 };
```

До змінних з типом об'єднання можна застосовувати всі ті ж самі операції, що й до структурних змінних: присвоєння (тобто копіювання цілого об'єднання), визначення адреси змінної, виділення окремих елементів (полів) об'єднання.

Для звертання до елементів об'єднання використовують такі ж синтаксичні конструкції на основі операцій "крапка" . та "стрілка" ->, як і в разі звертання до елементів структур:

ім'я_змінної_об'єднання . ім'я_поля

вказівник_на_об'єднання -> ім'я_поля

*(* вказівник_на_об'єднання) . ім'я_поля*

Наприклад, якщо виконати присвоєння

```
z.ch='A';
```

то у поле ch змінної z, оголошеної вище, буде занесено код символа 'A' (65). Тим самим наймолодші байти полів a та x набудуть значення 65. У разі наступного присвоєння:

```
(&z)->a = 0x3c32;          /* те ж саме, що й  z.a=0x3c32 */
```

зміняться два молодші байти об'єднання z. Тепер значенням виразу z.ch буде символ з шістнадцятковим кодом 0x32, тобто символ '2'.

З попередніх прикладів видно, що застосування об'єднань дає змогу маніпулювати окремими частинами даних та опрацьовувати їх нестандартними способами. Зокрема, через поле z.ch можна перевірити значення наймолодшого байта змінної z.x, що має тип double, а через звертання до поля z.a – змінити два молодші байти цієї змінної:

```
if (z.ch == 0)             /* перевірка наймолодшого байта */
   z.a = . . . ;           /* зміна двох молодших байтів */
```

Поля об'єднань можуть мати довільний тип – простий чи складений. Зокрема, поле об'єднання може бути структурою, вкладеним об'єднанням або вказівником на структуру чи об'єднання тощо. Водночас об'єднання можуть бути елементами масиву або складовими частинами структур чи інших об'єднань.

У наведеній нижче програмі об'єднання використовується для виділення складових частин довгого цілого числа number. Для цього number оголошено першим полем об'єднання un. Друге поле цього об'єднання є структурою struct longint_parts (її перейменовано в LIP), яка складається з двох двобайтових полів: поле low накладається на молодшу половину значення number, а поле high – на старшу половину цього числа. Третє поле об'єднання un – поле byte є масивом з чотирьох однобайтових значень, воно призначене для окремого звертання до кожного байта числа number.

```
/*****************************************************************/
/*  Застосування об'єднання для виділення частин довгого цілого  */
/*****************************************************************/
#include <stdio.h>
int main(void)
{
   typedef struct longint_parts {    /* шаблон структури, призначеної */
             unsigned int low, high;       /* для виділення молодшої */
           } LIP;                  /* і старшої половин довгого цілого */
   union number_parts {                          /* шаблон об'єднання */
      long number;                          /* число, що розглядається */
      LIP lip;                              /* структура половин числа */
      unsigned char bytes[4];                    /* масив байтів числа */
   } un = { 0x1a2b3c4dl };                        /* змінна об'єднання */
   int i;
```

```
  printf ("\nДовге ціле - %#0.10lx\n", un.number);
  printf ("Молодше слово - %#0.6x\t Старше слово - %#0.6x\n",
          un.lip.low, un.lip.high);
  puts ("\t Байти:");
  for ( i=0; i<4; i++)
    printf ("%#08.4x", (unsigned)un.bytes[i]);
  return 0;
}
```

Результат виконання:

```
Довге ціле - 0x1a2b3c4d
Молодше слово - 0x3c4d      Старше слово - 0x1a2b
          Байти:
0x4d    0x3c    0x2b    0x1a
```

У заголовному файлі <dos.h> системи програмування Borland C оголошено об'єднання union REGS, призначене для звертання з програми до системних регістрів і прапорців центрального мікропроцесора. Це об'єднання використовують як параметр функцій, що реалізують виклики системних переривань для керування роботою комп'ютера на апаратному рівні.

10.4. Поля бітів

Поля бітів (їх називають також *бітові поля*) – це особливий вид елементів структур та об'єднань. Такі поля складаються з послідовних бітів, записаних у дане цілого типу.

Поля бітів застосовують, коли необхідно мати доступ до окремих бітів всередині байта або слова даних. Найчастіше така потреба виникає у процесі керування апаратними засобами комп'ютера, наприклад, у разі звертання до регістрів відеоадаптера. Іншим поширеним застосуванням бітових полів є стиснення даних для компактного збереження інформації, насамперед у випадках, коли дані мають булів тип ("так" або "ні"), тому для їх збереження достатньо одного біта.

Оголошення поля бітів виконується через синтаксичну конструкцію:

тип_поля ім'я_бітового_поля : розмір_поля;

де *тип_поля* може бути тільки типом int з допоміжними специфікаторами unsigned або signed; *ім'я_бітового_поля* – звичайний ідентифікатор, а *розмір_поля* – ціле число, яке задає ширину поля в бітах.

Проілюструємо сказане практичним прикладом. Коли виведення інформації виконується в текстовому режимі роботи відеоадаптера, то для кожного символа, що висвітлюється на екрані, в оперативній пам'яті виділяються два байти: в перший байт заноситься ASCII-код символа, а в другий – його атрибути. Байт атрибутів має три складові частини (рис. 10.2): чотири молодші біти займає колір символа (він може набувати значень від 0 до 15), три наступні біти визначають колір фону даного символа (його

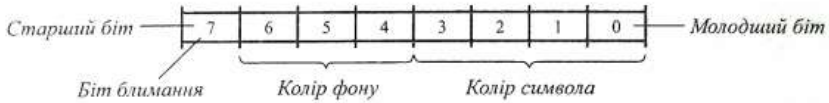


Рис. 10.2. Структура байта атрибутів

можливі значення від 0 до 7), а найстарший біт керує блиманням символа. Сформуємо відповідну структуру, елементами якої будуть три бітові поля:

```
struct text_attr{
   int  textcol:4;                /* колір символа */
   int  bkcol:3;                  /* колір фону */
   int  blink:1;                  /* мерехтіння */
} attr;
```

Якщо тепер виконати присвоєння:

```
attr.bkcol = 1;
attr.textcol = 15;
attr.blink = 1;
```

і пов'язати змінну `attr` зі значеннями атрибутів певного символа екрану, то даний символ буде відображений білим кольором на темно-синьому фоні і блиматиме. Щоб змінити кольори символа та фону на інверсні, виконаємо відповідні зміни у полях `attr`:

```
attr.textcol = ~attr.textcol;
attr.bkcol = ~attr.bkcol;
```

Зазначимо, що для опрацювання окремих бітів можна також використовувати поразрядні операції, розглянуті в розділі 4: накладання маски, зсування, побітове XOR тощо. Використання полів бітів робить програму більш зрозумілою та чіткою, водночас зменшуючи ризик помилки.

Структури та об'єднання, сформовані з полів бітів, здебільшого використовують як складові частини інших зовнішніх структур або об'єднань. З бітових полів не можна формувати масивів. Поля бітів не мають адрес.

Порядок розташування полів бітів, а також інші технічні питання, зокрема, чи може поле переходити через межу машинного слова, залежать від конкретної апаратної реалізації. Тому програми, в яких використовуються поля бітів, не належать до машинонезалежних. У програмах, відкомпільованих для MS DOS, поля бітів розташовуються послідовно від молодших адрес оперативної пам'яті до старших, а в межах байта – від молодших бітів до старших.

На завершення розглянемо програму, яка демонструє використання полів бітів для шифрування цілих беззнакових чисел, що мають тип `unsigned char`. Шифрування виконується шляхом переставлення місцями трьох молодших і трьох старших бітів двійкового коду числа. Для цього використано змінну `n`, що має тип об'єднання `union bit_fields_demo`, яке складається з трьох полів. У перше поле цього об'єд-

нання – numb записується число, яке шифрується. Друге поле cb, яке накладається на numb, є структурою зі шаблоном codebits, що складається з трьох бітових полів. Перше і останнє поле, кожне з яких має ширину 3 біти, переставляються місцями (тим самим змінюючи значення numb), середні 2 біти не використовуються. Третє поле об'єднання bit теж є структурою, що містить вісім однобітових полів. Цю структуру введено в об'єднання, щоб забезпечити доступ до кожного біта внутрішнього коду числа numb. У процесі роботи програми на екран спочатку виводяться десяткове і шістнадцяткове значення заданого числа та його двійковий код, а потім – відповідні значення результату перестановки бітів.

```
/*****************************************************/
/*  Застосування бітових полів для шифрування чисел  */
/*****************************************************/

#include <stdio.h>

int main(void)
{
  union bit_fields_demo {
     unsigned char numb;              /* число, що має бути зашифрованим */
     struct codebits {                /* перша структура бітових полів */
        unsigned int lbt:3;           /* три молодші біти */
        unsigned int mbt:2;
        unsigned int hbt:3;           /* три старші біти */
     } cb;
     struct bits {                    /* друга структура бітових полів */
        unsigned int b0:1, b1:1, b2:1, b3:1,b4:1, b5:1, b6:1, b7:1;
     } bit;
  } n = {167};
     unsigned int temp;               /* змінна для переставлення */
   printf ("\n3адане число - %d  ( %#0.4x -> ", n.numb, n.numb);
   printf ("%2d%2d%2d%2d%2d%2d%2d%2d )\n", n.bit.b7, n.bit.b6,
          n.bit.b5, n.bit.b4, n.bit.b3, n.bit.b2, n.bit.b1, n.bit.b0);
    /* Переставлення бітових полів */
   temp = n.cb.lbt;
   n.cb.lbt = n.cb.hbt;
   n.cb.hbt = temp;
   printf("\n3ашифроване число - %d  ( %#0.4x -> ",n.numb, n.numb);
   printf("%2d%2d%2d%2d%2d%2d%2d%2d )\n", n.bit.b7, n.bit.b6,
          n.bit.b5, n.bit.b4, n.bit.b3, n.bit.b2, n.bit.b1, n.bit.b0);
   return 0;
}
```

Приклад виконання:

```
Задане число - 167  ( 0xa7 ->  1 0 1 0 0 1 1 1 )
Зашифроване число - 229  ( 0xe5 ->  1 1 1 0 0 1 0 1 )
```

Запитання та завдання для самоконтролю

1. У чому особливість структурного типу даних? Що може бути полем структури?
2. Яке призначення шаблону структури? Як він оголошується? Що таке тег структури?
3. Оголосіть шаблон структури, яка описує книгу і містить поля: *<Автор>*, *<Найменування>*, *<Видавництво>*, *<Рік видання>*, *<Кількість сторінок>*.
4. Використовуючи шаблон структури з п. 3, оголосіть змінну `my_fav_book` та проініціалізуйте її даними про свою улюблену книгу.
5. Як визначити обсяг оперативної пам'яті, яку займає структурна змінна?
6. Які операції можна виконувати над структурними змінними?
7. Як можна звернутись до поля структури? У яких випадках застосовують операцію "крапка", а в яких – операцію "стрілка"? Наведіть приклади, використовуючи змінну `my_fav_book` і відповідний вказівник, що містить адресу цієї змінної.
8. За допомогою декларації `typedef` надайте шаблону структури з п. 3 ім'я `BOOK`.
9. Чим відрізняються об'єднання від структур? Яке призначення об'єднань?
10. У програмі оголошено й проініціалізовано об'єднання та відповідний вказівник:

```
union compose {
   struct twoparts {
      int fst, sec;
   } stp;
   unsigned long r;
} uc = { 0, 1 }, *pt = &uc;
```

Вкажіть значення наступних виразів:

1) `uc.stp.sec`
2) `uc.r`
3) `pt->r * 2`
4) `(* pt).stp.fst > 0`

11. Що називають полем бітів? Де застосовують бітові поля?

Запрограмуйте подані нижче задачі, використовуючи різні форми звертання до полів структур і об'єднань

12. Задано набір структур, кожна з яких задає координати (x, y, z) просторової точки. Визначити дві точки, віддаль між якими найбільша.
13. З клавіатури ввести послідовність структур, в які занести дані про хід передплати періодичних видань: *<Найменування видання>*, *<Індекс>*, *<Організація>*, *<Кількість примірників>*. Сформувати масив із введених даних, визначити і надрукувати сумарну кількість передплачених видань по кожному з найменувань.
14. Задано масив довгих цілих чисел. Необхідно зашифрувати його елементи, використовуючи таку схему: перший (наймолодший) байт числа переставляється місцями з третім, а другий – з найстаршим (четвертим). Вивести результат шифрування, а потім виконати дешифрування. Для реалізації процедур шифрування/дешифрування чисел треба застосувати відповідне об'єднання.

Розділ 11

ФУНКЦІЇ

У цьому розділі:

- Означення та загальна характеристика функцій
- Структура функцій: заголовок і тіло функції, тип значення, яке повертає функція, ім'я функції, оголошення формальних параметрів, внутрішні змінні та оператори функції, способи завершення функцій
- Звертання до функцій, фактичні параметри у викликах функцій
- Прототипи користувацьких і бібліотечних функцій
- Узгодження та взаємодія фактичних і формальних параметрів, передавання у функцію значень параметрів, звертання за адресами фактичних параметрів, порядок обчислення значень фактичних параметрів
- `inline`-функції
- Особливість опрацювання масивів і символьних рядків
- Робота з параметрами командного рядка
- Багатовимірні масиви як параметри функцій
- Способи опрацювання структур у функціях
- Вказівники на функції: оголошення, звертання до функцій через вказівники; вказівник на функцію як параметр іншої функції та вказівник на функцію як значення функції; використання масиву вказівників на функції
- Рекурсивні функції: порівняння рекурсивних та ітераційних обчислювальних процесів, рекурсивні алгоритми, хвостова та зворотна рекурсія, приклади характерних рекурсивних функцій
- Створення функцій з неоголошеними параметрами: безпосереднє звертання до неоголошених параметрів та використання макрозасобів `<stdarg.h>` для роботи з неоголошеними параметрами

Основними складовими компонентами кожної С-програми є функції, оператори яких реалізують алгоритм розв'язування задачі. *Функціями* називають самостійні, логічно завершені фрагменти програм, що мають власне ім'я та призначені для виконання певних заданих дій, останньою з яких може бути повернення деякого значення. Функції формують окремий тип даних мови С. Використання функцій дає змогу

структуризувати програму: поділити складні обчислювальні і/або інформаційні процеси на окремі складові частини, виділити основні кроки в алгоритмі розв'язування задачі, а вже потім розкрити деталі їх програмної реалізації в окремих функціях (це називають *програмуванням зверху вниз*). Завдяки функціям вдається уникнути багаторазового запису одних і тих самих дій та скористатись тим, що вже зроблено раніше, в тому числі бібліотекою системи програмування. Функції полегшують процес програмування, уможливлюють його розпаралелення, спрощують пошук помилок у програмі та внесення доповнень чи розширень, роблять програму виразнішою та лаконічною.

Характерною рисою (можна сказати, ідеологією) С-програм є те, що вони складаються із більшої кількості коротких функцій, а не з декількох великих.

Всі функції, включаючи функцію `main()`, яка обов'язково входить до складу кожної С-програми, рівноправні. Особливість функції `main()` у тому, що вона розпочинає роботу програми, а її завершення означає кінець виконання усієї програми. Саме в функції `main()` найчастіше вказують основні кроки алгоритму. В усьому іншому `main()` не відрізняється від решти функцій. Кожна функція С-програми записується окремо, вкладати функції одна в одну не можна.

Мова С забезпечує ефективний і простий механізм роботи з функціями. Кожну функцію програми або групу функцій можна відкомпілювати автономно і зберігати в окремому obj-файлі. Об'єктні коди відкомпільованих функцій долучаються до складу виконавчого коду програми (exe-файла) на етапі редагування зв'язків (компонування).

11.1. Структура функцій

Згідно зі стандартом мови С функція повинна описуватись синтаксичною конструкцією такої форми:

```
тип_значення_функції  ім'я_функції (оголошення параметрів)
{
   оголошення внутрішніх змінних
   оператори тіла функції
}
```

Перший рядок опису функції називається *заголовком функції*, а частина в фігурних дужках {...} формує *тіло* функції. Зупинимось на особливостях окремих конструктивних частин опису функцій.

Тип значення функції та ім'я функції. У заголовку функції першим вказується тип значення, яке повертає функція в точку її виклику після завершення виконання. Тип значення функції може бути довільним допустимим для мови С типом (у тому числі задекларованим через `typedef` користувацьким типом), крім масиву та функції. Зокрема, функція може повертати значення кожного з арифметичних типів (`char`, `int`, `double` тощо), вказівник на довільний базовий тип (включаючи вказівники на масиви та вказівники на функції), а також цілі структури й об'єднання. Особливим

типом значення функції є тип `void`, який вказує, що функція взагалі не повертає значення, – такі функції є аналогами процедур в інших мовах програмування.

Стандарт С-89 дозволяв опускати тип значення в описах функцій. У таких випадках за правилами замовчування вважалось, що значення, яке повертає функція, має тип `int`. Це спрощення критикували багато програмістів. У стандарті мови С-99 "правило неявного `int`" скасовано (його немає і в С++). Тепер у кожному описі та оголошенні функції тип її значення треба вказувати явно.

Ім'я функції призначене для ідентифікації даної функції, воно формується за правилами запису ідентифікаторів мови С. Для наочності програми дуже важливо, щоб імена функцій були змістовними і розкривали призначення відповідних функцій.

Список параметрів функції. Після імені функції завжди вказують круглі дужки, в яких оголошується список параметрів функції – їх називають *формальними* параметрами або *аргументами* функції. Функція може мати порожній список параметрів, але круглі дужки `()`, які є ознакою функції, вказуються обов'язково. Звернемо увагу, що в програмах мовою С відсутність параметрів функції, як правило, позначають не порожнім списком, а ключовим словом `void`, наприклад:

```
int ReadNumber (void)          /* функція без параметрів */
{
    . . .                      /* тіло функції */
}
```

Через слово `void` компілятор отримує інформацію, що виклик `ReadNumber()` повинен здійснюватись без вказання *фактичних* параметрів, тобто тих, що задаються у звертанні до функції. Повністю порожній (без `void`) список формальних параметрів у описі функції є вказівкою компілятору мови С не перевіряти відповідність фактичних і формальних параметрів у викликах даної функції.

У мові С++ правила дещо відмінні: порожній список однозначно вказує, що функція не використовує параметрів (так само, як це робить ключове слово `void` у списку параметрів С-функцій).

Якщо ж функція використовує параметри, то кожен з них оголошується у списку параметрів із окремим зазначенням типу (навіть коли типи послідовних параметрів збігаються), одне оголошення відділяється від іншого комою. Наприклад, функція

```
/*  Виведення значень трьох дійсних коренів  */
void Print3 (double x1, double x2, double x3)
{
   printf ("\n\t Три корені:\n x1 = %1.3lf      x2 = %1.3lf"
           "      x3 = %1.3lf \n", x1, x2, x3);
}
```

має три параметри (аргументи): `x1`, `x2` та `x3` з однаковим типом `double`, який вказується окремо для кожного з них. Типи параметрів функції можуть бути довільними типами мови С (особливості оголошення та опрацювання у функціях даних різних типів розглянемо далі).

Внутрішні змінні функції. Тіло функції, записане в фігурних дужках {}, реалізує дії, які повинна виконати ця функція. Здебільшого для роботи функції необхідні додаткові внутрішні змінні, їх ще називають *локальними* змінними. Такі змінні оголошуються на початку тіла функції перед першим оператором.

У програмах мовою C++, а також за стандартом C-99 оголошення змінних можна виконувати і в операторній частині функції (між операторами), але перед першим звертанням до цієї змінної.

Для прикладу наведемо функцію `Pow()`, призначену для піднесення заданого дійсного аргументу до цілого степеня. Функція використовує внутрішню змінну `prod`.

```
/*  Піднесення дійсного числа до цілого степеня  */
double Pow (double base, int n)
{
     double prod;
   for ( prod = 1.0;  n > 0;  n-- )
      prod *= base;                    /* добуток чисел */
   return prod;
}
```

Областю дії внутрішніх змінних функції та її формальних параметрів є область тіла даної функції. Такі змінні створюються на час виконання функції і стають невизначеними після її завершення. Тому звертатись до внутрішніх змінних функції з інших функцій не можна (питання області дії та часу існування локальних і глобальних змінних детально розглянемо в розділі 12).

Способи завершення функції. Функція завершує свою роботу, коли виконано всі оператори її тіла та досягнуто кінця функції або коли зустрінеться оператор `return`. Зупинимось на особливостях різних варіантів завершення роботи функції.

Якщо тип значення функції `void`, як у випадку `Print3()`, то це означає, що функція не повертає значення, тому оператор `return` для неї не обов'язковий. Проте, в разі необхідності, `return` можна застосувати для переривання роботи функції.

```
/*  Приклад переривання функції main()  */
void main (void)
{
   double d;
       . . .                           /* попередня частина програми */
    if (d<0) {
       puts ("Корені відсутні");
       return;                         /* завершення програми */
    }
       . . .                           /* наступні дії програми */
}
```

Коли ж функція повертає певне значення, тобто її тип відмінний від `void`, то це значення має бути передане через вираз, записаний в операторі `return` (прикладом

може слугувати описана вище функція `Pow()`). Значення виразу перетворюється до типу, заданого в заголовку функції, і заноситься у буфер обміну, звідки його можна отримати у тій точці програми, з якої викликають дану функцію.

Стандарт С не вважає помилкою, якщо функція, що має відмінний від `void` тип, завершується без кінцевого оператора `return` (компілятор Borland C виводить відповідне попередження). Проте значенням такої функції буде "сміття", тому треба уникати невизначених варіантів завершення функції. Ще одне зауваження: стандарт С-89 дозволяв використовувати у функціях, які за оголошенням повертають певне значення, оператор `return` без виразу, що теж спричиняло невизначеність результату. Новий стандарт С-99 скасував усі невизначеності. За цим стандартом вимагається, щоб функція обов'язково повертала через оператор `return` конкретне значення.

11.2. Виклик функції. Прототипи функцій

Виклик функції. Оператори тіла функції виконуються тільки тоді, коли здійснюється звертання до даної функції (це називають *викликом* функції). Виклик функції позначають операцією "круглі дужки" – `()`:

ім'я_функції (*список фактичних_параметрів*)

тут *список фактичних_параметрів* – це послідовність виразів, кожен з яких задає значення відповідного формального параметра, вказаного в описі даної функції.

Фактичні і формальні параметри повинні бути обов'язково узгодженими за кількістю, порядком розташування і типами (детальніше питання взаємодії фактичних і формальних параметрів розглянемо в наступних параграфах).

Виклик функції, в оголошенні якої вказано тип `void`, є окремим оператором програми (подібно до виклику процедур у мові Pascal). Наприклад, до описаної раніше функції `Print3()` можна звернутись так:

```
Print3 ( 3*x, y, z/y );
```

Мова С дозволяє також автономно (тобто через окремий оператор) викликати функції, які повертають певне значення – у цьому випадку значення, яке повертає функція, ігнорується. Прикладом можуть слугувати виклики бібліотечної функції `printf()`, яка повертає значення з типом `int`, але переважно це значення не використовують.

Коли функція повертає певне значення, то воно передається через спеціальний буфер у те місце програми, з якого викликали дану функцію. Значення, повернене функцією, можна використовувати у всіх виразах як звичайний операнд, тип якого збігається з типом значення функції.

Наведемо приклад обчислення значення виразу $r=(x^3-y^4)/2$ через звертання до функції `Pow()`, оголошеної вище:

```
r = (Pow(x, 3) - Pow(y, 4)) / 2;
```

Прототипи функцій. Компілятор С контролює правильність звертання до функції та застосування значення, яке вона повертає, тому опис функції повинен передувати звертанню до неї. Однак розташувати функції програми в потрібному порядку (так,

щоб кожну функцію, яка викликає інші функції, записати після тих функцій, які вона викликає) складно, а часто й зовсім неможливо (зокрема у разі перехресних викликів між функціями). Оскільки кожна функція є автономною програмною одиницею, то вкладення функцій не допускається. Крім цього, бажано, щоб першими в програмі були записані ті функції, які розкривають алгоритм розв'язування задачі, а функції, які деталізують окремі кроки, розміщувались після них. У великих програмах найкраще першою вказувати функцію `main()`.

Довільний порядок запису функцій у С-програмах забезпечується використанням т. зв. прототипів функцій. *Прототип* функції відтворює рядок її заголовка і закінчується знаком крапка з комою `;`, тобто має таку форму:

тип_значення_функції ім'я_функції (*оголошення типів_параметрів*) ;

Прототипи функцій треба вказувати перед першим звертанням до цих функцій. Найчастіше всі прототипи або їх більшу частину записують на початку програми. Можна також вказувати прототипи безпосередньо перед викликаючою функцією. Оскільки прототип функції надає повну інформацію про параметри та значення, яке повертає функція, то компілятор може перевірити правильність звертання до функцій, опис яких у тексті програми розташовано нижче.

У списку параметрів прототипу функції можна вказувати тільки типи параметрів, опускаючи їх імена. Наприклад, прототип функції `Print3()`, яка використовує три дійсні параметри (вона розглядалась раніше), можна записати наступним чином:

```
void Print3 ( double, double, double );
```

Дозволено також вказувати в прототипі імена параметрів, відмінні від тих, що використані в описі функції, приміром, оголосити прототип функції `Print3()` так:

```
void Print3 ( double q, double r, double u );
```

Наведемо приклад програми, яка виводить на екран результат звертання до функції `Pow()`, що підносить задане дійсне число до цілого степеня. Першою у програмі записано функцію `main()`, а функцію `Pow()` розміщено після неї. Тому перед `main()` оголошено прототип функції `Pow()`. Імена параметрів у прототипі відрізняються від тих, що використані в описі `Pow()`.

```
/*******************************************/
/*  Обчислення цілого степеня дійсного числа  */
/*******************************************/
#include <stdio.h>
double Pow (double b, int p);          /* прототип функції */
int main(void)
{
     double numb = 14.18;              /* число */
     int k = 9;                        /* показник степеня */
   printf ("\n\t%10.2lf ^ %d = %1.3le\n", numb, k, Pow(numb,k));
   return 0;
}
```

```
double Pow (double base, int n)      /* функція піднесення до степеня */
{
    double prod;
   for (prod = 1.0;  n > 0;  n--)          /* цикл обчислення добутку */
      prod *= base;
   return prod;
}
```

Результат виконання:

```
14.18 ^ 9 = 2.318e+10
```

Стандарт мови С дозволяє записувати порожній список параметрів у прототипі функції:

тип_значення_функції ім'я_функції ( );

Але практика показала, що таких оголошень прототипів слід уникати, оскільки в цьому випадку компілятор не отримує інформації про кількість і склад формальних параметрів функції. Тому він не може проконтролювати правильність запису фактичних параметрів у викликах функції та в разі потреби перетворити їх до типів відповідних формальних параметрів. Це може стати причиною небезпечних помилок у роботі функції. Наприклад, якщо прототип функції `Print3()` оголосити без параметрів:

```
void Print3();
```

то наступний виклик функції:

```
Print3(5, 48, 3.7);
```

спричинить виведення неправильних результатів, бо перші два фактичні параметри цілі числа, а прототип не вказує компілятору, що їх треба перетворити до типу `double`.

Прототипи бібліотечних функцій. Практично кожна С-програма використовує у своїй роботі ті чи інші бібліотечні функції. Це функції, тексти яких попередньо відкомпільовані, а їх об'єктні коди зберігаються у стандартній або користувацькій бібліотеці. Прототипи стандартних бібліотечних функцій С поділені на окремі групи за призначенням функцій та записані у спеціальних заголовних файлах із розширенням *.h. Наприклад, у файлі stdio.h зібрані прототипи всіх функцій високорівневого буферизованого введення-виведення даних, у файлі conio.h – прототипи функцій обміну даними через консольні пристрої, у файлі string.h – прототипи функцій для роботи з ділянками пам'яті й символьними рядками, а в файлі stdlib.h – прототипи функцій різного призначення тощо.

У разі використання тих чи інших бібліотечних функцій необхідно підключити до тексту програми заголовні файли, в яких записані прототипи цих функцій. Наприклад, директива

```
#include <time.h>
```

на етапі препроцесування вставить у текст програми вміст стандартного заголовного файла time.h, що дасть змогу викликати з програми бібліотечні функції визначення дати та часу. Ознайомитись із прототипами та призначенням функцій бібліотеки Borland C, які оголошені в названих вище заголовних файлах, можна в Додатку 2.

11.3. Взаємодія фактичних і формальних параметрів функції

Коли відбувається звертання до функції, то всі її формальні параметри отримують значення відповідних фактичних параметрів, заданих у виклику цієї функції. Тому кількість фактичних параметрів повинна строго відповідати кількості формальних параметрів, вказаних в оголошенні функції. Виняток становлять тільки спеціальні функції зі змінною кількістю параметрів, які розглянемо пізніше. Самі ж фактичні параметри можуть бути довільними виразами, але значення кожного з них має бути сумісним з типом відповідного формального параметра (сумісність означає, що значення виразу може бути перетворене до типу формального параметра).

Механізм взаємодії фактичних і формальних параметрів у процесі виконання функції наступний:

1) послідовно обчислюються значення виразів, заданих у виклику функції, тобто значення її фактичних параметрів;
2) значення кожного з фактичних параметрів перетворюється до типу відповідного формального параметра, оголошеного в описі або прототипі функції (нагадаємо: якщо список параметрів у прототипі функції порожній, то компілятор C не виконує перетворення типів фактичних параметрів, а компілятор C++ вважає, що дана функція не використовує параметрів);
3) перетворення типів фактичних параметрів виконується згідно з правилами узгодження типів в операціях присвоєння (див. параграф 4.9); додатково всі фактичні та формальні параметри з типами `char` і `short` (знакові та беззнакові) розширюються до типу `int`, а параметри з типом `float` – до типу `double`;
4) отримані значення фактичних параметрів послідовно одне за одним заносяться у стек-область оперативної пам'яті, виділену для параметрів даної функції;
5) формальні параметри функції набувають значень, занесених у стек; реально кожен формальний параметр отримує адресу початку ділянки, виділеної для значення відповідного фактичного параметра, і розглядає вміст цієї ділянки як значення з типом, вказаним в оголошенні цього формального параметра;
6) виконуються оператори тіла функції, які опрацьовують значення, що їх набули формальні параметри функції;
7) якщо функція повертає певне значення, то в буфер обміну заноситься значення виразу, який записано в операторі `return`, перетворене до типу, вказаного в оголошенні функції;
8) після завершення роботи функції пам'ять, яку було виділено для параметрів і внутрішніх змінних цієї функції, звільняється.

Передавання значень параметрів. З описаної схеми взаємодії фактичних і формальних параметрів зрозуміло, що в тілі функції опрацьовуються копії значень фактичних параметрів, занесені в стек, а значення самих фактичних параметрів при цьому не змінюються. Це важливий практичний момент, який слід обов'язково враховувати, організовуючи роботу функцій.

Нехай функцію `Max()`, яка визначає найбільше з трьох заданих дійсних значень, у програмі описано так:

```
/*  Функція визначення найбільшого з трьох дійсних чисел  */
double  Max (double x, double y, double z)
{
   x = (x > y) ? x : y;
   return (x > z) ? x : z;
}
```

У разі наступного оголошення змінних програми і звертання до функції:

```
   double u, max;
max = Max(u, 50, u*u /25);
```

формальний параметр `x` функції `Max()` отримає значення змінної `u`, параметр `y` – значення константи `50.0`, а параметр `z` – значення виразу `u*u/25`. Всі три значення будуть перетворені до типу `double`. Такий взаємозв'язок параметрів називають *звертанням за значенням* (*call by value*).

Ще раз звернемо увагу: якщо для оголошення функції використати прототип з порожнім списком параметрів

```
double  Max();
```

то компілятор C не буде контролювати і перетворювати параметри функції. Отже константа `50` буде записана в стек як значення з типом `int`, а не `double`, тому параметр `y` і всі наступні формальні параметри отримають неправильні значення.

У тілі функції `Max()` значення параметра `x` змінюється, проте ці зміни не впливають на значення змінної `u`, заданої як фактичний параметр у виклику функції.

Класичним зразком неправильного звертання за значеннями параметрів є приклад функції, яка мала би міняти місцями значення двох своїх цілочислових аргументів. Якщо цю функцію записати так:

```
/*  Помилковий варіант функції обміну  */
void  Swap1(int a, int b)
{
     int d = a;
   a = b;     b = d;
}
```

і викликати, щоб поміняти місцями значення двох змінних:

```
   int  alpha = 25, beta = 18;
Swap1(alpha, beta);
```

то значення `alpha` і `beta` не зміняться. Хоча в тілі `Swap1()` формальні параметри `a` і `b` обмінюються значеннями, це не впливає на значення фактичних параметрів.

Звертання за адресами параметрів. Щоб функція могла змінити значення змінної, треба передати у функцію адресу цієї змінної. Тоді з функції можна буде звертатись до змінної за вказаною адресою та оперувати безпосередньо з її значенням, а не з копією,

занесеною в стек. Очевидно, що відповідний формальний параметр функції повинен бути вказівником, базовий тип якого збігається з типом змінної. Таку організацію взаємозв'язку параметрів, коли у функцію передаються адреси змінних, називають *звертанням через посилання* (*call by reference*). Пригадаємо стандартну функцію введення `scanf()`. Всі її параметри, що вказуються у списку введення, вказівники – вони задають адреси, за якими функція `scanf()` записує зчитані з клавіатури та перетворені у внутрішню форму числові й символьні значення.

Повернемось до функції обміну. Щоб переставити місцями значення двох змінних, функція повинна оперувати з їхніми адресами, тобто формальні параметри функції повинні бути вказівниками на змінні, значення яких переставляються:

```
/*  Правильний варіант функції обміну  */
void  Swap2 (int *pa, int *pb)
{
     int d=*pa;
   *pa=*pb;     *pb=d;
}
```

Проілюструємо використання цієї функції прикладом програми, яка переставляє у зворотному порядку елементи масиву `ar` з індексами від `n1` до `n2`, де `n1` і `n2` – два випадкових числа.

```
/***********************************/
/*  Переставлення елементів масиву  */
/***********************************/
#include <stdio.h>
#include <stdlib.h>
int Swap (int* p1, int* p2);        /* прототип функції переставлення */
int main(void)
{
     int ar[] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
     int n1, n2, k;
     int *p, *q;
   k = sizeof(ar)/sizeof(int);          /* кількість елементів масиву */
   randomize();
   n1 = random(k);       /* індекс першого елемента, що переставляється */
   n2 = random(k);       /* індекс останнього переставленого елемента   */
   if (n1 > n2)
      Swap(&n1, &n2);
   for (p = ar+n1, q = ar+n2;  p < q;  p++, q--)
      Swap(p, q);                       /* переставлення елементів масиву */
   printf("\nРезультат переставлення елементів %d - %d:\n",
          n1+1, n2+1);
   for (p = ar;  p < ar+k;  p++)
      printf("%d  ", *p);
   return 0;
}
```

```
int Swap (int *pa, int *pb)                    /* функція переставлення */
{
     int d=*pa;
   *pa=*pb;     *pb=d;
}
```

Приклад виконання:

```
Результат переставлення елементів  5 - 16:
1  2  3  4  16  15  14  13  12  11  10  9  8  7  6  5  17  18
```

Перший раз у програмі функцію `Exchange()` викликаємо для впорядкування значень `n1` і `n2`, передаючи у функцію адреси цих змінних. Далі циклічно використовуємо `Exchange()` для переставлення місцями двох відповідних елементів масиву. На цей раз параметрами виклику функції є вказівники на елементи, які переставляються.

Порядок обчислення фактичних параметрів. Часто важливим є порядок обчислення фактичних параметрів у реалізації виклику функції, зокрема у випадках, коли функція містить т. зв. неоголошені параметри (ці функції будемо розглядати в параграфі 11.11), або коли значення параметрів функції взаємопов'язані, як у наступному прикладі виклику функції `Print3()`, яку описано раніше:

```
    double x=5.0;
Print3 ( 2*x, ++x, x );
```

Інкремент другого параметра `++x` змінить значення змінної `x` і вплине на значення першого або третього параметра залежно від встановленого порядку обчислення параметрів. Тому найкраще не використовувати взаємозалежних параметрів у звертаннях до функцій, оскільки результат реалізації таких функцій буде визначатись особливостями конкретного компілятора.

У Borland C встановлено порядок, за яким першим обчислюється і заноситься у стек значення останнього параметра функції, потім значення передостаннього параметра і т. д. Таку послідовність опрацювання параметрів називають C-порядком передавання аргументів у функцію. Наприклад, якщо записаний вище виклик `Print3()` реалізувати в середовищі Borland C, то результат виконання функції буде таким:

```
Три корені:
x1 = 12.000       x2 = 6.00       x3 = 5.000
```

Останній фактичний параметр виклику функції `(x)` опрацьовувався першим, тому він зберіг значення `5.0`, обчислення другого параметра `(++x)` не тільки передало у функцію значення `6.0`, але й змінило значення зміної `x`, що відповідно вплинуло на значення першого параметра `(2*x)`, який обчислювався останнім.

Встановлений у Borland C порядок обчислення значень фактичних параметрів функцій відмінний від традиційного, за яким першим опрацьовується значення першого параметра, потім другого і т.д. – саме такий порядок встановлено для підпрограм мови Pascal. Це ускладнює взаємодію C-програм з функціями, написаними іншими мовами програмування, і навпаки, C-функцій з програмами, розробленими для інших мов.

Компілятор Borland C підтримує два спеціальні специфікатори: `cdecl` та `pascal` (обидва є зарезервованими словами). Специфікатор `cdecl` (*C-declaration*) вказує, що параметри функції будуть опрацьовуватись за порядком, властивим мові C, тобто починаючи від останнього. Цей специфікатор використовується, зокрема, в оголошеннях бібліотечних функцій Borland C. Наприклад, прототип функції `window()` у заголовному файлі conio.h оголошено так:

```
void cdecl  window (int left, int top, int right, int bottom);
```

Альтернативний спосіб опрацювання параметрів – від першого до останнього – задається специфікатором `pascal`. Специфікатор `pascal` застосовують у випадках, коли функції, написані мовою C, будуть використовуватись у Pascal-програмах, а також в оголошеннях прототипів функцій, які написані та відкомпільовані як функції мови Pascal.

Змінити стандартний порядок роботи з параметрами функції можна також через відповідну опцію меню інтегрованого середовища Borland C: Options / Compiler / Entry / Exit Code... / Calling Convention.

11.4. `inline`-функції

Функції, оголошення яких починається ключовим словом `inline`, називають *вбудованими* функціями. `inline`-функції є нововведенням, реалізованим стандартом C-99, хоча програмісти вже давно застосовують такі функції, оскільки вони підтримуються мовою C++.

Ключове слово `inline`, записане перед заголовком функції, є вказівкою для компілятора, що виклики даної функції повинні бути максимально швидкими. Здебільшого це означає, що компілятор "вбудує" код функції у місце її виклику (подібно до макропідстановок з параметрами). Звісно, таке вбудовування може збільшити обсяг коду програми, оскільки кожен виклик функції буде замінено її тілом. Тому як `inline` доцільно оголошувати невеликі функції, які викликаються багатократно і мають критичний вплив на час виконання програми.

Використання вбудованих функцій дає змогу, з одного боку, дотримуватись принципів функціональної структуризації програми, а з іншого – забезпечити максимальну швидкодію реалізації таких функцій. Виграш у часі виконання `inline`-функцій порівняно зі звичайними функціями досягається завдяки тому, що вбудований код не потребує передавання параметрів і значення функції, а також збереження в стеку поточних станів системних регістрів. Проте слід пам'ятати, що `inline` не є обов'язковою директивою, тобто компілятор може і не вбудовувати дану функцію, а використати інші засоби оптимізації швидкодії чи взагалі проігнорувати вказівку `inline`.

Для прикладу наведемо `inline`-функцію `Cube()`, що обчислює значення куба заданого аргументу дійсного типу.

```
/*  Вбудована функція піднесення до куба  */
inline  double Cube (double x)
{
   return x * x * x ;
}
```

Якщо тепер звернутись до записаної функції для обчислення виразу:

```
Cube(a+b) / (Cube(a) - Cube(b))
```

то цілком вірогідно, що компілятор поміняє виклики функції на вбудовування її тіла, тобто вираз набуде вигляду:

```
(a+b) * (a+b) * (a+b) / (a * a * a - b * b * b)
```

Хоча реальна підстановка може бути організована й дещо по-іншому.

11.5. Масиви та символьні рядки як параметри функцій

Є певні особливості у використанні масивів, а тим самим і символьних рядків як параметрів функцій. Якщо у функцію для опрацювання передається ім'я масиву, то на відміну від змінних інших типів не створюється копія масиву, а передається адреса його першого елемента. Тому в процесі виконання функції здійснюється робота з тим масивом, який задано через відповідний фактичний параметр.

11.5.1. Масиви – параметри функцій

Використовують дві форми оголошення формального параметра функції, який повинен відповідати масиву:

- *тип_елементів ім'я_масиву*`[]`, наприклад: `int vect [];`
- *тип_елементів* `*` *ім'я_вказівника*, наприклад: `int *pv`.

Обидві форми рівнозначні та оголошують формальний параметр, що є вказівником на початок масиву. Перша форма візуально підкреслює, що функція опрацьовуватиме масив, а друга зазначає, що для роботи з елементами масиву використовується вказівник.

Проілюструємо обидві форми оголошення масивів як параметрів функцій та різні способи звертання до елементів фактичних масивів прикладами функцій `Average1()`, `Average2()` та `Average3()`, кожна з яких обчислює середнє значення `n` послідовних елементів масиву дійсних чисел.

У функції `Average1()` параметр `mas[]` явно оголошений як масив, а для звертання до його елементів використано класичну індексну форму.

```
/*  Обчислення середнього значення елементів масиву. Варіант 1 */
double Average1 (double mas[], int n)
{
     int i;
     double sum = 0;
   for (i = 0;  i < n;  i++)
      sum += mas[i];
   return sum / n;
}
```

У функції `Average2()` формальний параметр `pa`, через який задається масив, оголошено як вказівник, а для звертання до елементів масиву використано операцію розадресації. Звернемо увагу, що параметр `pa` є звичайним вказівником, тому для звертання до елементів масиву не обов'язково вводити додатковий вказівник – можна просто пересувати `pa`, інкрементуючи його значення `pa++`.

```
/*  Обчислення середнього значення елементів масиву. Варіант 2  */
double Average2 (double *pa, int n)
{
     int i;
     double * pn, sum = 0;
   for (pn = pa+n;  pa < pn;  pa++)
      sum += *pa;
   return sum / n;
}
```

Щоб підтвердити рівноправність обох форм оголошення параметрів-масивів, наведемо ще один варіант функції визначення середнього значення.

```
/*  Обчислення середнього значення елементів масиву. Варіант 3  */
double Average3 (double m[], int n)
{
     double  sum = * m,  * end = m+n;
   while (++m < end)
      sum += * m;
   return sum / n;
}
```

Хоча параметр `m` оголошено в заголовку функції `Average3()` як масив `m[]`, у тілі функції його використано як звичайний вказівник, який просувається по елементах масиву операцією `++m`. Підкреслимо, що формальний параметр функції, навіть оголошений як масив, є звичайним, а не константним вказівником на початок масиву, тому його значення можна змінювати в тілі функції.

Звертання до кожної з трьох описаних вище функцій буде однаковим: треба передати у функцію адресу початку масиву та кількість елементів, що мають бути просумовані. Наприклад, якщо в програмі оголошено масив і три змінні:

```
double arr[150], ar1, ar2, ar3;
```

та виконано присвоєння

```
ar1 = Average1(arr, 150);
```

то в функцію `Average1()` передається адреса першого елемента масиву `arr`. У результаті виконання функції змінна `ar1` отримає середнє значення всіх 150-ти елементів цього масиву.

Можна використати описані функції для сумування лише певної частини елементів масиву. У разі наступних викликів:

```
ar2 = Average2(&arr[10], 40);
```

```
ar3 = Average3(arr+60, 75);
```

змінна `ar2` набуде середнього значення елементів масиву `arr`, які мають індекси від `10` до `49`, а змінна `ar3` – середнього значення 75-ти послідовних елементів масиву, починаючи від елемента з індексом `60`.

11.5.2. Символьні рядки – параметри функцій

Все сказане про параметри-масиви цілком стосується і тих параметрів функцій, які є символьними рядками. Їх також можна оголошувати як масиви або як вказівники – остання форма на практиці використовується частіше.

Опрацювання символьних рядків у функціях. Процеси опрацювання символьних рядків будуть ефективними, якщо для звертання до символів рядка застосовувати вказівники. Розглянемо для прикладу функцію, яка в заданому реченні замінює всі малі літери української та латинської абетки на відповідні великі літери.

```
/*  Функція запису символьного рядка великими літерами  */
#include <string.h>
#include <ctype.h>
char* UkrStrUp (char* str)
{
     char *small = "абвгґдеєжзиіїйклмнопрстуфхцчшщьюя";
     char *capit = "АБВГҐДЕЄЖЗИІЇЙКЛМНОПРСТУФХЦЧШЩЬЮЯ";
     char *ps, *pl;
   for (pl=str;  *pl!='\0';  pl++) {
      ps=strchr(small,*pl);
      if (ps!=NULL )          /* якщо знайдено малу українську літеру */
         *pl=capit[ps-small];                /* замінюємо її великою */
      else
         *pl=toupper(*pl);                 /* заміна латинських літер */
   }
   return str;
}
```

У тілі функції `UkrStrUp()` використано два константні символьні рядки з наборами малих і відповідних великих літер української абетки. Якщо літера речення є малою літерою української абетки (входить у рядок `small`), то замінюємо її на відповідну за порядковим номером велику літеру (номер літери визначаємо через різницю вказівників `ps-small`). Інакше, якщо це мала латинська літера, то підставляємо замість неї велику, використовуючи стандартну бібліотечну функцію `toupper()`, прототип якої записаний у файлі `<ctype.h>`.

Функція `UkrStrUp()` повертає вказівник на початок перетвореного рядка. Такий тип поверненого значення здебільшого застосовують у функціях, які змінюють вміст символьного рядка. Зокрема, саме таке значення повертає функція введення символьних рядків `gets()` та більшість бібліотечних функцій опрацювання символьних рядків, оголошених у `<string.h>`. Хоча адреса, яку повертає функція, відома (це адреса

першого символа рядка, що опрацьовувався), все ж наявність поверненого значення дає змогу використовувати цю функцію як параметр інших функцій, які працюватимуть із перетвореним символьним рядком. Наприклад, якщо в програмі оголошено масив-рядок:

```
char anon[]="Увага! Увага!  Attention!!";
```

то для виведення цього рядка великими літерами можна застосувати функцію `puts()` з параметром, який є викликом функції `UkrStrUp()`:

```
puts ( UkrStrUp(anon));
```

Як результат виклику `puts()` буде надруковано:

```
УВАГА! УВАГА!  ATTENTION!!
```

Якби функція `UkrStrUp()` не повертала адреси перетвореного рядка, тобто якби тип її значення був `void`:

```
void  UkrStrUp (char * str)
{
   . . .                  /* тіло функції */
   return;
}
```

то виведення перетвореного рядка потребувало би двох операторів виклику функцій:

```
UkrStrUp (anon);
puts (anon);
```

Щоб показати деякі інші практичні прийоми, які доцільно використовувати в процесах опрацювання символьних рядків, наведемо програму, яка виділяє зі символьного рядка всі цілі числа, обчислює і друкує їх значення (реалізація подібної задачі на основі стандартної бібліотечної функції `strtol()` вже розглядалась у параграфі 9.4). У поданій нижче програмі застосовано дві користувацькі функції `FindNumber()` і `GetNumber()`. Функція `FindNumber()` шукає цифри в заданому символьному рядку і повертає адресу першої знайденої цифри або `NULL`, якщо цифр у рядку немає. Функція `GetNumber()` обчислює і повертає значення довгого цілого числа, записаного в символьному рядку, адресу початку якого задає перший параметр цієї функції. Додатково через другий параметр `GetNumber()` передає адресу символа, наступного за числом. Цей символ стає початком рядка, в якому слід шукати наступне ціле число. У функції `main()` обидві функції викликаються циклічно, поки з рядка не будуть виділені всі числа:

```
while ((pnum=FindNumber(pnum))!=NULL)              /* поки є числа */
   printf("%ld\n", GetNumber(pnum, &pnum));        /* виводимо їх  */
```

Звернемо увагу на кілька моментів. У виразі умови оператора `while` ітераційно змінюється значення вказівника `pnum`: `pnum = FindNumber(pnum)`. Перед викликом функції `pnum` вказує на символ, з якого треба починати пошук цифри, а після завершення роботи `FindNumber()` вказівнику `pnum` присвоюється адреса першої знайденої цифри. Ця адреса передається через перший параметр у функцію `GetNumber()` для

формування числа, починаючи з вказаної цифри: `GetNumber(pnum, &pnum)`. Через другий параметр у `GetNumber()` передається адреса вказівника `pnum`, щоб у тілі функції записати в цей вказівник адресу частини рядка, яка ще не опрацьовувалась, і через `pnum` передати її в наступний виклик функції `FindNumber()`.

Для перетворення рядка в число у функції `GetNumber()` використано наступну схему: 12345 => (((1×10+2)×10+3)×10+4)×10+5. Вираз `*pn-'0'` визначає числове значення поточної цифри як різницю кодів символів.

```
/*************************************************/
/*  Виділення всіх цілих чисел із символьного рядка  */
/*************************************************/
#include <stdio.h>
#include <ctype.h>
char* FindNumber (char* s)                    /* функція пошуку цифри */
{
   while (!isdigit(*s) && *s != '\0')              /* пошук цифри */
      s++;
   if (*s == '\0')                              /* якщо цифр немає */
      return NULL;
   else
      return s;
}
long GetNumber (char* pn, char** next)        /* функція обчислення  */
{                                             /* значення long-числа */
     long numb=0;
   while (isdigit(*pn)) {                        /* формування числа */
       numb=numb*10+(*pn-'0');
       pn++;
   }
   *next=pn;       /* збереження адреси символа, наступного за числом */
   return numb;
}
int main (void)
{
     char str[]="Вхідні дані: 45000, 366, 9705 (100750).";
     char *pnum=str;
     int k=0;
   while ((pnum=FindNumber(pnum))!=NULL)
      printf ("\nЧисло %d => %ld ", ++k, GetNumber(pnum,&pnum));
   return 0;
}
```

Результат виконання:

```
Число 1 => 45000
Число 2 => 366
Число 3 => 9705
Число 4 => 100750
```

Створення у функції нового символьного рядка. Нагадаємо, що значенням, яке повертає функція, не може бути масив чи символьний рядок. Проте в багатьох практичних задачах потрібно, щоб у функціях створювались нові масиви або стрінги. Розглянемо приклад функції, яка повинна виділяти зі заданого речення перше слово.

```
/*  Виділення першого слова речення - помилковий варіант  */
#include <ctype.h>
char* WrongFirstWord (char *s)
{
     char word[20], *w=word;
   while (isspace(*s))              /* пошук початку слова */
      s++;
   while (*s!=' ' && *s!=',' && *s!=':' && *s!='.' && *s!='\0')
      *w++=*s++;                    /* копіювання літер до кінця слова */
   *w='\0';
   return word;
}
```

Наведена функція відтворює помилку, яку часто допускають програмісти-початківці в задачах подібного типу. Тому в разі виклику функції:

```
printf("\t Перше слово - %s\n", WrongFirstWord("Один, два, три"));
```

слово, виведене на екран, буде не `Один`, як можна було очікувати, а складатиметься із непередбачуваного набору символів, наприклад, такого:

```
        Перше слово - K*
```

У тілі функції `WrongFirstWord()` перше слово речення `"Один, два, три"` виділяється правильно і функція повертає адресу початку цього слова. Все ж помилка в тому, що виділене слово заноситься у рядок `word`, який є внутрішньою змінною функції. Як вже зазначалось, після завершення роботи кожної функції пам'ять, яку займали параметри і внутрішні змінні цієї функції, вважається вільною. У процесі виконання іншої функції (у нашому прикладі це `printf()`) ділянка, яку раніше займав масив `word`, може бути заповнена даними цієї функції. Тому за адресою, яку повертає `WrongFirstWord()`, будуть зчитуватися дані тієї функції, яка виконується у цей момент.

Щоб сформувати у функції новий масив чи символьний рядок, треба: 1) виділити місце для цього масиву (рядка) у викликаючій функції; 2) включити до списку параметрів функції, яка формує масив (рядок), адресу виділеної для нього ділянки.

Альтернативний варіант – створення нового масиву чи символьного рядка в динамічній пам'яті. Після завершення виклику функції такий масив (рядок) залишається у виділеній ділянці динамічної пам'яті і може опрацьовуватись іншими функціями, треба тільки коректно повернути з функції його адресу. Ще одним способом є використання глобальних масивів і символьних рядків. Проте такі масиви (рядки) не захищені від випадкових змін, оскільки до них можна звертатись з кожної точки програми.

У наступній програмі подано приклад правильної версії функції, яка виділяє та повертає перше слово заданого речення.

```
/*********************************************************/
/*  Виділення першого слова речення - правильний варіант  */
/*********************************************************/
#include <stdio.h>
#include <ctype.h>
char* FirstWord (char* s, char* word)
{
     char* w=word;
   while (isspace(*s))   s++;                  /* пошук початку слова */
   while (*s!=' ' && *s!=',' && *s!=':' && *s!='.' && *s!='\0')
      *w++=*s++;                              /* копіювання першого слова */
   *w='\0';
   return word;
}
int main (void)
{
     char str[]="\"Орфей\" - переможець конкурсу";          /* рядок */
     char wrd1[20];                                  /* масив для слова */
   printf ("\n Перше слово: %s \n", FirstWord(str, wrd1));
   return 0;
}
```

Результат виконання:

```
Перше слово: "Орфей"
```

Використання кваліфікатора `const` в оголошеннях параметрів. Якщо аргументом функції є адреса деякого фактичного параметра, то така функція оперує безпосередньо з цим параметром і може змінювати його значення. У багатьох випадках така зміна небажана або взагалі недопустима. Щоб захистити значення фактичного параметра від змін, спричинених помилковим записом у тілі функції якихось даних за адресою цього параметра, треба в оголошенні відповідного формального аргументу вказати кваліфікатор `const`. Тоді компілятор зафіксує кожну спробу запису даних у константний параметр, а також спробу присвоєння адреси цього параметра внутрішньому вказівнику, що оголошений без кваліфікатора `const`. Крім цього, ключове слово `const` обв'язкове в оголошеннях тих формальних параметрів-вказівників, яким будуть присвоюватись адреси фактичних параметрів, оголошених з кваліфікатором `const`.

Якщо ж формальний параметр-вказівник оголосити з кваліфікатором `const`, записаним безпосередньо перед іменем вказівника, то компілятор стежитиме, щоб у тілі функції значення цього вказівника не змінювалось.

Таким чином, щоб зробити функцію `FirstWord()` універсальною і захистити її параметри від випадкових змін, оголошення параметрів слід записати так:

```
char* FirstWord (const char* s, char* const word)
{
     . . .                  /* тіло функції */
}
```

11.6. Робота з параметрами командного рядка

Дотепер ми використовували функцію `main()` як функцію без параметрів, позначаючи це ключовим словом `void` у списку параметрів функції. Проте функції `main()`, як й іншим функціям С-програм, можна передавати необхідну інформацію. Дані, які передаються у функцію `main()`, називають *параметрами (аргументами) командного рядка*. Їх записують у командному рядку операційної системи після імені виконавчого файла (exe-файла) програми, функція `main()` якої буде опрацьовувати ці дані. Наприклад, якщо програму, exe-код якої зберігається у файлі comline.exe, запустити на виконання з операційного середовища, заповнивши командний рядок так:

```
D:\LRPR\comline.exe temporary output 168
```

то функція `main()` цієї програми отримає три параметри: `"temporary"`, `"output"`, та `"168"`. Відразу зазначимо, що функція `main()` розглядає дані командного рядка як окремі стрінги (рядки символів), навіть якщо вони записані у формі чисел (параметр 168 у наведеному прикладі). Загалом параметри командного рядка є словами, що відокремлюються між собою символами пробілу або табуляції.

Операційне середовище MS DOS дозволяє записувати параметри, що складаються з декількох слів. Такі параметри беруть у лапки, наприклад:

```
D:\LRPR\comline.exe book "Red and Black" 10
```

Проте не всі середовища підтримують таку форму запису параметрів.

Щоб функція `main()` могла опрацьовувати дані, записані в командному рядку, треба в її оголошенні вказати два формальні параметри:

```
int main (int argc, char * argv[])
{
    . . .                  /* тіло функції main() */
}
```

Перший параметр `main()`, який прийнято іменувати `argc` (скорочено від слів *argument count*), цілочисловий. Він отримує від операційної системи значення, що дорівнює кількості параметрів, записаних у командному рядку. Другий параметр `argv` (від слів *argument vector*) – це вказівник на перший елемент внутрішнього масиву вказівників, які зберігають адреси стрінгів, переданих з командного рядка (рис. 11.1).

Рис. 11.1. Схема звертання до параметрів командного рядка

Імена параметрів `argc` і `argv` не обов'язкові, але програмісти здебільшого використовують саме їх.

Пояснимо детальніше призначення параметрів функції `main()`. Якщо програма запускається на виконання без додаткових даних у командному рядку, то параметр `argc` отримує значення `1`. Це вказує, що в командному рядку записано тільки ім'я exe-файла програми, яке може включати шлях до цього файла. Якщо ж у командному рядку додатково записані параметри виконання програми, то значення `argc` дорівнюватиме загальній кількості параметрів командного рядка. Таким чином, для наведених вище прикладів `argc` отримає значення `4`.

Параметри командного рядка заносяться в оперативну пам'ять як окремі символьні рядки. Для роботи з ними створюється спеціальний масив вказівників, кожен елемент якого зберігає адресу відповідного параметра (див. рис. 11.1). Останній елемент масиву заповнюється порожнім вказівником `NULL`. Адреса першого байта створеного масиву вказівників присвоюється формальному параметру `argv`.

У тілі функції `main()` параметри `argc` і `argv` можна опрацьовувати так само, як опрацьовують формальні параметри відповідних типів у всіх інших функціях. Наведемо приклад програми, яка виводить на екран параметри, передані їй через командний рядок операційного середовища.

```
/*****************************************/
/*  Виведення параметрів командного рядка  */
/*****************************************/
#include <stdio.h>
int main (int argc, char *  argv[])
{
    int i;
  if (argc==1)                                  /* параметри відсутні */
    printf("У командному рядку немає параметрів\n");
  else {
    printf("У програму %s передано такі параметри:\n",argv[0]);
    for (i=1;  i<argc;  i++)
      printf("%d-й =>  %s\n", i, argv[i]);   /* виведення параметрів */
  }
  return 0;
}
```

Результат виконання для описаного вище прикладу даних командного рядка:

```
У програму D:\LRPR\comline.exe передано такі параметри:
1-й =>  temporary
2-й =>  output
3-й =>  168
```

Якщо програма запускається на виконання не з операційного середовища, а з інтегрованого середовища програмування Borland C, то дані, які мають бути передані в функцію `main()`, потрібно попередньо занести в рядок введення командних параметрів *Arguments* у відповідному вікні діалогу, яке викликається через пункт меню Run.

Оскільки argv формальний параметр, якому передається адреса першого елемента масиву вказівників, то його часто оголошують у main() наступним чином: char ** argv, тобто як вказівник на вказівник на дані з типом char. Запишемо приклад ще однієї програми, яка шукає перший параметр командного рядка, що починається знаком мінус '-'.

```
/*******************************/
/*  Пошук від'ємного параметра  */
/*******************************/
#include <stdio.h>
int  main (int argc, char* argv[])
{
   for (++argv;  *argv!=NULL;  argv++)
      if ((*argv)[0]=='-')              /* перевірка першого символа */
         break;
   if (*argv==NULL)
      puts("\nНемає від\'ємних параметрів");
   else
      puts(*argv);
   return 0;
}
```

Зауважимо, що в цій програмі argv використовується як звичайний вказівник для просування по елементах масиву. Операція ++argv на початку циклу for виконується, щоб відразу перенести argv з елемента масиву, який вказує на ім'я програми, на елемент, що зберігає адресу першого параметра командного рядка. Перший символ кожного параметра виділяємо виразом (*argv)[0]. Круглі дужки () у цьому виразі необхідні з огляду на старшинство операції [] щодо операції розадресації *. Ознакою кінця списку параметрів слугує порожній вказівник NULL. Оскільки значення NULL дорівнює 0, то можна скоротити запис циклу перегляду параметрів:

```
/*  Компактний варіант циклу  */
   while (*++argv)                  /* замість argv++ і *argv!=NULL */
      if (**argv=='-') {            /* замість (*argv)[0]=='-' */
         puts(*argv);     return 0;
      }
   puts("\Немає від\'ємних параметрів");
```

Якщо певний параметр, переданий через командний рядок, треба опрацьовувати в main() як число, то його необхідно перевести в числову форму. Для цього можна застосувати відповідні бібліотечні або користувацькі функції, наприклад:

```
   int k;
k = atoi(argv[3]);
```

Якщо список параметрів є таким, як на рис. 11.1, то змінна k отримає значення 168.

Параметри командного рядка часто застосовують для передавання програмі імен файлів, які мають опрацьовуватись у цій програмі, чи певних ключових параметрів, які

визначатимуть роботу програми тощо. Наприклад, наступна програма буде виконуватись тільки за умови, що в командному рядку вказано правильне слово-пароль:

```
/*****************************/
/*  Захист програми паролем  */
/*****************************/
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
     char *password="слово-пароль";
   if (argc!=2 || strcmp(argv[1], password)!=0)
      return 0;                  /* без пароля програма не виконується */
   . . .                         /* тіло програми */
}
```

Функція `main()` може мати третій параметр, який за типом збігається з `argv` і звичайно оголошується так: `char * enrp[]`. За змістом він є вказівником на початок масиву вказівників, що містять адреси символьних рядків, у які занесено інформацію про т. зв. *оточення програми*, тобто про поточний стан операційного середовища.

На завершення ще раз нагадаємо, що стандартно функція `main()` має повертати значення цілого типу. Це значення передається у програму, яка запустила на виконання дану користувацьку програму (здебільшого це одна із системних програм). Якщо в `main()` не реалізовано повернення завершального значення, то більшість компіляторів (серед них і компілятор Borland C) автоматично вважають, що функція `main()` повертає значення `0`. Проте для мобільності програми необхідно, щоб `main()` явно повертала цілочисловий результат завершення програми.

11.7. Багатовимірні масиви як параметри функцій

Як відомо, мова C розглядає багатовимірні масиви як масиви, елементами яких є підмасиви меншої вимірності. Тому, якщо багатовимірний масив треба передати в функцію для опрацювання, то відповідний формальний параметр функції має бути вказівником на перший підмасив цього багатовимірного масиву. Як і у випадку одновимірних масивів, для оголошення формальних параметрів, що відповідають багатовимірним масивам, можна застосовувати дві форми:

- *тип_елементів ім'я_масиву*`[][`*к-сть_елементів_1*`]...[`*к-сть_елементів_n*`];`
- *тип_елементів* `(*` *ім'я_вказівника*`)[`*к-сть_елементів_1*`]...[`*к-сть_елементів_n*`].`

В обох формах оголошення треба обов'язково вказувати кількість елементів для кожного виміру масиву, крім першого. Багатовимірний масив зберігається в оперативній пам'яті як послідовність підмасивів. Тому для знаходження адреси певного елемента (чи порядкового номера елемента в масиві) за значеннями його індексів треба знати розмірність кожного виміру цього масиву.

Вказівникова форма оголошення параметра, що є багатовимірним масивом, вимагає використання круглих дужок, оскільки операція * має нижчий пріоритет, ніж операція []. Без круглих дужок відповідний параметр буде масивом вказівників на *тип_елементів*, а не вказівником на підмасив заданого багатовимірного масиву.

Проілюструємо різні форми звертання до багатовимірних масивів у функціях прикладом програми, яка виводить на екран значення та координати найбільшого елемента заданої матриці дійсних чисел. У програмі наведено три варіанти функцій: `MaxElement1()`, `MaxElement2()` та `MaxElement3()`, кожна з яких повертає значення максимального елемента матриці, а координати цього елемента записує за адресами, заданими двома параметрами-вказівниками. У функціях реалізовано різні способи оголошення параметра-матриці та звертання до її елементів. Детальніші пояснення наведено після програми.

```
/*****************************************************************/
/*  Визначення найбільшого елемента матриці та його координат  */
/*****************************************************************/
#include <stdio.h>
#define N 25
/* Перший варіант функції визначення найбільшого елемента матриці */
double  MaxElement1 (double matr[][N], int nr, int nc,
                     int *mr, int *mc)
{
  /*       Параметри функції:
       matr - матриця дійсних чисел;
       N - розмірність рядка матриці в оперативній пам'яті;
       nr, nc - кількість рядків і стовпців, що опрацьовуються;
       mr, mc - адреси для запису координат максимального елемента     */
     int i, j;
     double max=matr[0][0];              /* приймаємо за максимальний */
   *mr=0;    *mc=0;                     /* перший елемент матриці */
   for (i=0;  i<nr;  i++)
      for (j=0;  j<nc;  j++)
         if (matr[i][j]>max) {
            max=matr[i][j];
            *mr=i;    *mc=j;           /* запис координат max */
         }
   return max;
}
/* Другий варіант функції визначення найбільшого елемента матриці */
double  MaxElement2 (double (*pm)[N], int nr, int nc,
                     int *mr, int *mc)
{
     int i, j;
     double max = **pm, *pel;
   *mr=0;    *mc=0;
```

```
   for (i=0;  i<nr;  i++, pm++)
      for (j=0, pel=*pm;  j<nc;  j++, pel++)
         if (*pel>max) {
            max=*pel;
            *mr=i;    *mc=j;
         }
   return max;
}
/* Третій варіант функції визначення найбільшого елемента матриці */
double  MaxElement3 (double *pr, int n, int nr, int nc,
                     int *mr, int *mc)
{
     double max, *pel=pr, *first=pr;
   *mr=0;    *mc=0;
   for (max=*pr;  nr;  nr--, pr+=n)
      for (pel=pr;  pel<pr+nc;  pel++)
         if (*pel>max) {
            max=*pel;
            *mr=(pr-first)/n;        /* номер рядка елемента max */
            *mc=pel-pr;              /* номер стовпця елемента max */
         }
   return max;
}
int  main (void)
{
    double m[10][N]={{12.4,1,3.8},{3,8,56.3},{7,55.6,8},{12,4.8,6}};
    double maxel;
    int r,c;
  maxel=MaxElement1(m, 4, 3, &r , &c);
  printf("\nФункція MaxElement1: max=%.1f (%d,%d)", maxel, r+1, c+1);
  maxel=MaxElement2(m, 4, 3, &r , &c);
  printf("\nФункція MaxElement2: max=%.1f (%d,%d)", maxel, r+1, c+1);
  maxel=MaxElement3(m[0], N, 4, 3, &r , &c);
  printf("\nФункція MaxElement3: max=%.1f (%d,%d)", maxel, r+1, c+1);
  return 0;
}
```

Результат виконання:

```
Функція MaxElement1:  max=56.3 (2,3)
Функція MaxElement2:  max=56.3 (2,3)
Функція MaxElement3:  max=56.3 (2,3)
```

У функції `MaxElement1()` використано явну форму оголошення параметра `matr` як матриці:

```
double matr[][N]                    /* параметр - матриця */
```

Звертання до елементів матриці в функції `MaxElement1()` здійснюється через індекси: `matr[i][j]`. З урахуванням розмірності рядка `N`, заданої в оголошенні матриці, індексна форма відповідає наступному звертанню (так інтерпретує її компілятор):

```
matr[i][j] ≡ *(*matr + i*N + j)
```

Тому оголошення параметра `matr` без вказання розмірності рядка матриці в оперативній пам'яті `N` було б хибним:

```
double matr[][]              /* хибне оголошення параметра-матриці */
```

У функції `MaxElement2()` параметр-матрицю оголошено як вказівник на масив з `N` дійсних чисел, тобто як вказівник на рядок матриці:

```
double (*pm)[N]              /* параметр - вказівник на рядок */
```

Збільшення цього вказівника в тілі циклу `pm++` пересуває його на наступний рядок матриці – значення `pm` збільшується на величину `N*sizeof(double)`. Для просування по елементах поточного рядка матриці введено внутрішній вказівник `pel`, який на початку кожного циклу ініціалізується адресою першого елемента рядка `pel=*pm`.

Три наступних оголошення параметра `pm` теж будуть хибними для даної задачі:

```
double **pm                  /* хибне оголошення параметра-матриці */
double *pm[N]
double (*pm)[]
```

У першому записі `pm` оголошено як вказівник на вказівник на дані з типом `double`, а не як вказівник на рядок матриці; у другому варіанті оголошення `pm` задано як масив з `N` вказівників; у третьому варіанті параметр `pm` оголошено як вказівник на масив, але не вказано розмірність цього масиву, тому операції адресної арифметики, зокрема `pm++`, для вказівника `pm` не визначені, а саме оголошення помилкове.

Остання функція `MaxElement3()` використовує параметр `pr`, який є вказівником на елементи матриці. Слід звернути увагу на відповідний фактичний параметр у виклику функції – він має бути адресою першого елемента матриці. Оскільки параметр `pr` не несе ніякої інформації про розмірність матриці, то, щоб коректно визначати адресу початку кожного рядка, до списку параметрів функції введено додатковий параметр `n`, який задає розмірність рядка матриці. Вираз `pr+=n` встановлює вказівник `pr` на перший елемент наступного рядка матриці. Цей варіант організації функції найбільш універсальний, бо дає змогу опрацьовувати матриці довільного розміру, а не тільки ті, які оголошено з розмірністю рядка `N`.

Як і в попередній функції, для звертання до елементів рядка в `MaxElement3()` використано додатковий вказівник `pel`. Координати максимального елемента обчислюються як різниця значень вказівників, щоб не вводити додаткових змінних.

Ще одне зауваження до наведеної програми. У функції `main()` друкуються результати звертання до кожної з функцій – спочатку виводиться значення максимального елемента, а потім його координати:

```
printf("\nФункція ...: max=%.1f (%d,%d)", maxel, r+1, c+1);
```

Якщо змінну `maxel` замінити викликом однієї з функцій, наприклад:

```
printf("\nФункція MaxElement1: max=%.1f (%d,%d)",
       MaxElement1(m,4,3,&r,&c), r+1, c+1);
```

то замість значень координат максимального елемента може виводитись "сміття". В середовищах, де фактичні параметри функцій опрацьовуються починаючи від останнього, значення виразів `c+1` і `r+1` обчислюються і заносяться у стек перед викликом `MaxElement1()`, тобто перед тим, як у змінні `r` та `c` буде записано координати максимального елемента матриці.

Стандарт С-99 надав додаткові можливості для роботи з масивами та структурами в функціях. Ці нововведення описано в Додатку 3.

11.8. Опрацювання структур у функціях

На відміну від масивів, структури можна копіювати та присвоювати, оскільки мова С інтерпретує їх як звичайні змінні, а не як вказівники. Тому можна передавати значення структури у функцію через відповідний формальний параметр або повертати структуру як значення результату виконання функції.

Якщо певна структура опрацьовується функціями програми, то шаблон такої структури повинен бути описаний на зовнішньому рівні, тобто перед функціями, які використовують даний тип структури.

Нехай в програмі оголошено шаблон структури, призначеної для збереження дати у формі: *день*, *найменування_місяця*, *рік*:

```
struct full_date {              /* шаблон структури, оголошений */
   int day;                     /* на зовнішньому рівні */
   char mon[10];
   int year;
};
```

Запишемо функцію, яка заповнює поля структури `struct full_date` даними поточної дати і повертає сформовану структуру. Щоб отримати інформацію про поточну дату, використаємо бібліотечну функцію `getdate()`, прототип якої оголошений у заголовному файлі `<dos.h>`. Ця функція формує структуру зі встановленим у `<dos.h>` шаблоном `struct date` (він складається з трьох цілочислових полів) і записує її у змінну, адреса якої задається єдиним параметром функції `getdate()`.

```
/*  Функція формування поточної дати  */
#include <dos.h>
struct full_date GetCurDate (void)
{
  struct date dat;                  /* структура, оголошена в <dos.h> */
  struct full_date curd;
  char * mon_name[] = { "Січень", "Лютий",  "Березень", "Квітень",
   /* масив назв */    "Травень", "Червень", "Липень", "Серпень",
   /* місяців    */    "Вересень", "Жовтень", "Листопад", "Грудень" };
```

```
    getdate (&dat);                 /* визначення поточної дати - <dos.h> */
    curd.day = dat.da_day;              /* заповнення полів змінної curd */
    strcpy (curd.mon, mon_name[dat.da_mon-1]);
    curd.year = dat.da_year;
    return curd;                                /* повернення структури-дати */
}
```

Значення, яке повертає функція `GetCurDate()`, можна присвоїти довільній змінній, що має тип `struct full_date`. Прикладом є наступний фрагмент програми, який заповнює змінну `today` даними поточної дати та виводить результат:

```
    struct full_date today;
  today = GetCurDate();
  printf("Сьогодні:  %s, %d - %d р.", today.mon, today.day, today.year);
```

На екран виводиться поточна дата у формі рядка:

```
Сьогодні:  Травень, 14 - 2004 р.
```

Способи передавання структур. Використовують різні способи передавання структур у функцію для опрацювання. Серед них:

- передавання цілих структур через відповідні параметри-структури;
- передавання адрес структур через параметри – вказівники на структури;
- передавання окремих полів структур.

Залежно від конкретного випадку, кожен з цих способів має свої переваги й недоліки.

Проілюструємо різні форми опрацювання структур у функціях на прикладі такої задачі. Задано масив структур, в яких зберігаються персональні дані членів наукового товариства (через `typedef` шаблону структур присвоєно ім'я `PDAT`). Одне з полів структури, а саме `workplace`, задає місце праці науковця:

```
typedef struct person_data {        /* шаблон структури з даними */
   . . .                            /* члена наукового товариства */
   char workplace[60];
} PDAT;
```

Нехай певна наукова установа змінила своє найменування. Треба внести відповідні зміни в масив структур. Розглянемо три функції, призначені для зміни поля `workplace`.

Перший варіант базується на передаванні у функцію значення всієї структури. Якщо поле `workplace` цієї структури збігається зі заданим найменуванням установи `oldname`, то воно замінюється новим найменуванням `newname`, інакше структура не змінюється. Функція повертає опрацьовану структуру.

```
/* Перший варіант функції зміни найменування місця праці */
#include <string.h>
PDAT  ChangeWorkPlace1 (PDAT member)
{
    char * oldname = " старе_найменування ";
    char * newname = " нове_найменування ";
```

```
    if (strcmp(member.workplace, oldname) == 0)
       strcpy(member.workplace, newname);
    return member;
}
```

Для внесення змін у всю базу даних функцію потрібно застосувати циклічно:

```
#define N 180
PDAT persondat[N];
     . . .                    /* заповнення даними масиву persondat[N] */
 for (i=0;  i<N;  i++)
    persondat[i]=ChangeWorkPlace1(persondat[i]);
```

Наведена версія реалізації поставленої задачі найбільш нераціональна, оскільки в процесі опрацювання кожної структури масиву тричі виконується її копіювання, навіть, якщо поля структури не змінюються взагалі.

Перше копіювання структури відбувається у момент виклику функції, коли елемент масиву `persondat[i]` переписується у формальний параметр `member`. Друга копія створюється у процесі виконання оператора

```
return member;
```

коли вміст структури `member` переписується у буфер обміну. Останнє копіювання виконує оператор, що присвоює елементу масиву `persondat[i]` значення, яке повертає функція `ChangeWorkPlace1()`.

Набагато ефективнішим є наступний варіант функції з використанням формального параметра `pmemb`, що є вказівником на структуру, яку необхідно опрацьовувати. У разі виклику функції `ChangeWorkPlace2()` цей параметр отримує адресу відповідної структури, тому в тілі функції відбуваються звертання безпосередньо до потрібних полів і не витрачається час на копіювання структур.

```
/* Другий варіант функції зміни найменування місця праці */
void  ChangeWorkPlace2 (PDAT * pmemb)
{
  char *oldname = "старе_найменування",  *newname = "нове_найменування";
    if (strcmp(pmemb->workplace, oldname) == 0)
      strcpy(pmemb->workplace, newname);
}
```

Приклад використання функції для внесення змін у весь масив:

```
for (i = 0;  i < N;  i++)
   ChangeWorkPlace2(persondat+i);          /* або &persondat[i] */
```

Третій варіант функції `ChangeWorkPlace3()` використовує формальний параметр `workname`, що безпосередньо задає адресу символьного рядка, який треба перевірити і в разі потреби змінити. Відповідним фактичним параметром у викликах цієї функції має бути адреса поля структури, в якому записано найменування місця праці.

```
/* Третій варіант функції зміни найменування місця праці */
char* ChangeWorkPlace3 (char* workname)
{
  char *oldname = " старе_найменування ", *newname = " нове_найменування ";
   if ( !strcmp(workname, oldname) )
      strcpy(workname, newname);
   return workname;
}
```

Приклад використання функції `ChangeWorkPlace3()`:

```
   PDAT *pdat;                /* вказівник на елементи масиву структур */
for (pdat = persondat;  pdat < persondat+N;  pdat++)
    ChangeWorkPlace3 (pdat -> workplace);
```

11.9. Вказівники на функції

Мова С розглядає функції як особливий тип даних – ім'я функції є константним вказівником на перший байт виконавчого коду даної функції. Значенням імені функції є адреса оперативної пам'яті, яка відповідає т. зв. *точці входу* даної функції. У разі виклику функції зчитується перша команда за цією адресою, а далі всі наступні команди, що реалізують дії функції. Таким чином, звертання за адресою точки входу функції фактично означає перехід на початок виконавчого коду даної функції. Адресу функції можна присвоїти відповідно оголошеному вказівнику та використовувати його для звертання до функції.

11.9.1. Оголошення вказівника на функцію. Звертання через вказівник

Щоб оголосити вказівник на функцію, використовують наступну синтаксичну конструкцію:

тип_значення_функції (* *ім'я_вказівника*) (*список типів_параметрів_функції*) ;

Операція `()` – "функція" має вищий пріоритет, ніж операція `*` – "вказівник", тому конструкцію `*` *ім'я_вказівника* необхідно охопити круглими дужками, щоб встановити правильний порядок інтерпретації оголошення. Без внутрішніх дужок дане оголошення було би прототипом функції, яка використовує відповідні параметри і повертає значення, яке є вказівником. У списку параметрів функції вказують тільки їх типи (без імен).

Ось приклад оголошення вказівника на функцію:

```
char* (*pfun)(char*, unsigned);
```

Тепер змінну `pfun` можна використовувати як вказівник на довільну функцію, що має два параметри: перший з типом `char*`, а другий з типом `unsigned`, і повертає вказівник на дані з типом `char`. Якщо в програмі оголошено дві функції:

```
char* FindWord (char* st, unsigned numb);
char* DeleteWords (char* sent, unsigned k);
```

то коректними будуть усі наступні присвоєння:

```
pfun = FindWord;          або     pfun = &FindWord;
pfun = DeleteWords;       або     pfun = &DeleteWords;
```

З наведених прикладів видно, що для присвоєння вказівнику адреси певної функції можна використовувати тільки її ім'я або вираз *& ім'я_функції*. Коли вказівник отримав адресу функції, його можна застосовувати для звертання до цієї функції. Наприклад, якщо значенням `pfun` є адреса функції `FindWord()`, то наступне звертання:

```
(*pfun)(str, 3);
```

рівнозначне відповідному виклику функції за її іменем

```
FindWord(str, 3);
```

Оскільки значення обох вказівників: константного `FindWord` і звичайного `pfun` однакові та задають адресу точки входу функції `FindWord()`, то для звертання до функції через вказівник можна використати також спрощену форму:

```
pfun(str, 3);
```

Хоча така форма виклику функції простіша у записі, проте вона не вказує явно, що `pfun` є вказівником, а не іменем функції. Тому для наочності програми краще застосовувати конструкцію з розадресованим вказівником на функцію.

Підкреслимо ще раз два важливі моменти в оголошеннях вказівників на функції. По-перше, необхідно використовувати внутрішні круглі дужки. Коли в програмі оголошено:

```
void (*pr1)(int);
void * pr2 (int);
```

то `pr1` є іменем вказівника на функцію, яка має один цілочисловий параметр і не повертає значення, а `pr2` – це ім'я функції, що має такий самий параметр і повертає значення, що має тип `void*`. По-друге, вказівнику на функцію можна присвоювати адреси тільки тих функцій, типи параметрів і тип значення яких повністю збігаються з типами, заданими в оголошенні цього вказівника. Тому наступне присвоєння

```
pr1 = pr2;                  /*  Невірно!  */
```

помилкове через невідповідність типів `void` та `void*`.

11.9.2. Вказівник на функцію як параметр функцій

Найчастіше вказівники на функції використовуються як формальні параметри у функціях вищого (в алгоритмічно-структурному розумінні) рівня. Це дає змогу створювати функції, які опрацьовують або використовують певний тип інших функцій без огляду на їх конкретні імена та внутрішнє наповнення.

Розглянемо функцію, призначену для обчислення середнього значення довільної математичної функції однієї змінної, аргумент і значення якої мають дійсний тип. Значення відповідної математичної функції повинні обчислюватися у K послідовних точках проміжку [x0, xk].

```
/*  Функція обчислення середнього значення математичної функції,
    заданої вказівником pf, на інтервалі [x0,xk]  */
#define K 100                    /* кількість точок аналізу функції */
double AverFunValue (double x0, double xk, double(*pf)(double))
{
     double x, dx, fsum = 0;
   dx = (xk-x0)/(K-1);                /* віддаль між точками аналізу */
   for (x = x0, xk += dx/10;  x < xk;  x += dx)
      fsum += (*pf)(x);               /* сумування значень функції */
   return  fsum / K;
}
```

Функція AverFunValue() має три параметри: перші два задають межі проміжку, а третій – pf вказує на математичну функцію, для якої обчислюється середнє значення. В оголошенні pf та в звертаннях до математичної функції використано операцію розадресації вказівника на функцію:

`double(*pf)(double)` – оголошення вказівника;

`(*pf)(x)` – звертання до функції.

В обох випадках можна було скористатися спрощеними формами:

`double  pf(double)` – для оголошення вказівника;

`pf(x)` – для звертання до функції.

Звернемо увагу на ще одну особливість програми: в заголовку циклу значення кінцевої межі проміжку xk дещо посувається вправо: xk += dx/10. Це зроблено для того, щоб через імовірне нагромадження похибки, викликаної багаторазовим додаванням дійсного значення кроку dx, не пропустити останню точку проміжку.

Проілюструємо використання функції AverFunValue() прикладом програми, яка обчислює середнє значення двох математичних функцій fun1() та fun2() на проміжку, заданому користувачем.

```
/*************************************************************/
/*  Обчислення середнього значення двох математичних функцій  */
/*************************************************************/
#include <stdio.h>
#include <math.h>
#define K 100
double fun1 (double);                          /* прототипи функцій */
double fun2 (double);
double AverFunValue (double, double, double(*)(double));
```

```
int main (void)
{
     double a, b;                                  /* межі проміжку */
   printf ("\nІнтервал => ");
   scanf ("%lf%lf", &a, &b);
   printf ("\n Середнє значення функції_1 => %+.4lf",
                                        AverFunValue(a, b, fun1));
   printf ("\n Середнє значення функції_2 => %+.4lf\n",
                                        AverFunValue(a, b, fun2));
   return 0;
}
/*  функція, що визначає середнє значення математичної функції,
    заданої вказівником pf, на інтервалі аналізу [x0,xk]              */
double AverFunValue (double x0, double xk, double(*pf)(double))
{
     double  x, dx, fsum=0;
   dx=(xk-x0)/(K-1);
   for (x=x0, xk+=dx/10;  x<xk;  x+=dx)
      fsum+=(*pf)(x);
   return fsum/K;
}
double fun1 (double x)            /* перша математична функція */
{
   return sin(2*x)-0.85*cos(x+1.4);
}
double fun2 (double x)            /* друга математична функція */
{
   if (x>=0)
      return 3*cos(1.5*x);
   else
      return cos(x)*cos(x);
}
```

Приклад виконання:

```
Інтервал =>  -0.6  2.7
Середнє значення функції_1 => +0.3418
Середнє значення функції_2 => -0.3252
```

У викликах функції `AverFunValue()`, що виконуються в `main()`, третім параметром вказується ім'я тієї математичної функції програми, для якої обчислюється середнє значення. Можна явно вказати, що передається саме адреса функції, записавши перед її іменем знак `&`. Два наступні виклики рівнозначні:

```
AverFunValue(a, b, fun1) ≡ AverFunValue(a, b, &fun1)
```

11.9.3. Функції, що повертають вказівник на функцію

Значення, яке повертає функція, не може бути ні масивом, ні функцією, оскільки ні масиви, ні функції не копіюються. Проте значення функції може бути довільною адресою, тому функція може повернути вказівник на функцію певного типу. Оголошення функції, значенням якої є адреса іншої функції (тобто вказівник на функцію), виконується через достатньо складну синтаксичну конструкцію:

тип_значення_функції_ФР (* *ім'я_БФ* (*список параметрів_БФ*)) (*список типів_параметрів_ФР*);

Абревіатурою *БФ* (базова функція) позначено функцію, яка оголошується, а *ФР* (функція результату) – функцію, вказівник на яку повертає базова функція. Як інтерпретувати такі непрості оголошення пояснимо далі на конкретних прикладах.

Першим наведемо приклад короткої функції `HigherValueFun()`, що вибирає з двох описаних вище математичних функцій `fun1()` та `fun2()` ту, середнє значення якої на проміжку $[-\pi/2, \pi]$ більше. Щоб вибрану функцію можна було використовувати в наступних діях програми, `HigherValueFun()` повинна повертати адресу цієї функції. Тому значенням функції `HigherValueFun()` буде вказівник (посилання) на вибрану математичну функцію з більшим середнім значенням.

```
/*  Функція, що визначає, яка з двох математичних функцій
          fun1() чи fun2() має більше середнє значення  */
#define Pi 3.14159265
double (* HigherValueFun(void))(double)
{
   if (AverFunValue(-Pi/2, Pi, fun1) > AverFunValue(-Pi/2, Pi, fun2))
     return fun1;
   else
      return fun2;
}
```

Зупинимось на заголовку наведеної функції. Як і будь-яке інше оголошення мови С, його треба починати читати від основного імені. В нашому прикладі це ідентифікатор `HigherValueFun`. Аналізуємо всю конструкцію у круглих дужках, які використано для того, щоб встановити потрібний порядок інтерпретації полів оголошення. Спочатку розглядаємо запис:

```
(* HigherValueFun(void))
```

Оскільки внутрішня операція `()` – "функція" має вище старшинство, ніж операція `*` – "вказівник", то робимо висновок, що `HigherValueFun` є іменем функції, яка не використовує параметрів `(void)` і повертає значення, яке є вказівником. Далі необхідно встановити базовий тип вказівника, який повертає функція `HigherValueFun()`. Щоб виділити цей тип, замінимо конструкцію, яку ми вже прочитали, одним іменем, наприклад, `F`. Тоді непрочитана частина заголовка функції буде такою:

```
double F(double)
```

Тепер виразно видно, що `F` є функцією, яка використовує один дійсний параметр і повертає значення, що має тип `double`. Знаючи інтерпретацію `F`, робимо загальний висновок, що `HigherValueFun()` – це функція без параметрів, яка повертає вказівник на іншу функцію, що використовує один дійсний параметр і повертає значення дійсного типу.

Тип функції, вказівник на яку повертає `HigherValueFun()`, збігається з типом функцій `fun1()` та `fun2()`. Це дає змогу після порівняння середніх значень обох функцій повернути адресу тієї з них, середнє значення якої на проміжку $[-\pi/2, \pi]$ більше. Якщо в програмі оголошено також вказівник `phf`:

```
double (*phf)(double);
```

то йому можна присвоїти адресу вибраної функції:

```
phf = HigherValueFun();
```

і надалі використовувати `phf` для звертання до математичної функції з більшим середнім значенням.

Доповнимо описану вище функцію `HigherValueFun()` так, щоб можна було вибирати функцію з більшим середнім значенням з двох довільних заданих функцій. Для цього в список параметрів функції вибору введемо два вказівники на математичні функції, які порівнюються.

```
/*  Функція, що повертає адресу математичної функції
    з більшим середнім значенням  */

#define Pi 3.14159265

double (* HigherAverValueFun (double (*f1)(double),
                              double (*f2)(double))) (double)
{
   if (AverFunValue(-Pi/2, Pi, f1) > AverFunValue(-Pi/2, Pi, f2))
      return f1;
   else
      return f2;
}
```

Прочитаємо заголовок наведеної функції. `HigherAverValueFun` є функцією, яка має два параметри `f1` та `f2`, кожен з яких є вказівником на функцію з одним аргументом дійсного типу, яка повертає значення з типом `double`. У свою чергу, функція `HigherAverValueFun()` повертає вказівник на функцію, що має один дійсний параметр і значення з типом `double`. Тобто типи обох параметрів і тип значення функції `HigherAverValueFun()` збігаються.

Щоб визначити, яка з математичних функцій `fun1()` чи `fun2()` має більше середнє значення на проміжку $[-\pi/2, \pi]$, можна виконати наступне звертання до функції `HigherAverValueFun()`:

```
phf = HigherAverValueFun(fun1, fun2);
```

Складні оголошення функцій можна набагато спростити, якщо попередньо використати декларацію `typedef` для іменування типів.

Нехай у програмі задекларовано:

```
typedef double FUNC (double x);     /* декларування типу функції */
```

Тоді тип `FUNC` буде визначати функцію, яка повертає значення з типом `double`, і використовує один параметр дійсного типу. Прототипи функцій, розглянутих в попередніх прикладах, і вказівник `phf` тепер можна оголосити так:

```
FUNC  fun1, fun2, * phf;
double  AverFunValue (double, double, FUNC);
FUNC *  HigherAverValueFun (FUNC *, FUNC *);
```

Зверніть увагу, наскільки спростилось і стало наочнішим оголошення функцій. Наведемо повний опис функції `HigherAverValueFun()`, щоб показати, що застосування типу `FUNC` не вплинуло на роботу з параметрами функцій.

```
FUNC * HigherAverValueFun (FUNC * f1, FUNC * f2)
{
   if (AverFunValue(-Pi/2, Pi, f1) > AverFunValue(-Pi/2, Pi, f2))
      return f1;
   else
      return f2;
}
```

11.9.4. Масиви вказівників на функцію

Такі масиви є ще одним поширеним застосуванням вказівників на функції. Синтаксична конструкція, що оголошує масив вказівників на функції, повинна мати таку форму:

тип_значення_функції (* *ім'я_масиву* [*кількість_елементів*]) (*список типів_параметрів_функції*) ;

Як і у випадку оголошення функцій, що повертають вказівник на функцію, дане оголошення потребує внутрішніх круглих дужок, щоб операція `*` – "вказівник" інтерпретувались перед операцією `()` – "функція".

Наступне оголошення створює масив вказівників на функції:

```
double (* farray[8])(double);
```

Такі оголошення читають, починаючи з імені змінної, за правилами, розглянутими раніше. Тобто `farray` – це масив з восьми елементів, кожен з яких є вказівником на функцію, що має один дійсний параметр і повертає значення з типом `double`.

Якщо в оголошенні масиву використати задекларований вище тип `FUNC`, то воно стане значно лаконічнішим:

```
FUNC * farray[8];
```

Масиви вказівників на функції опрацьовуються так само, як і звичайні масиви: їх можна ініціалізувати адресами відповідних функцій, до їхніх елементів можна звертатись через індекси, або через вказівники тощо. Наприклад, якщо виконати присвоєння

```
farray[0] = fun1;
farray[1] = fun2;
```

а потім обчислити значення наступного виразу і присвоїти його змінній `vf1f2`:

```
double vf1f2;
vf1f2 = (AverFunValue(0, 2, farray[0]) + AverFunValue(0,2, farray[1])) / 2;
```

то результат присвоєння дорівнюватиме півсумі середніх значень функцій `fun1()` та `fun2()` на проміжку [0, 2].

Основним застосуванням масивів вказівників на функції є створення користувацьких меню та організація т. зв. *таблиць переходів* (*jump-tables*). Щоб створити масив вказівників на функції, всі розділи меню або виконавчі блоки таблиць переходів оформляють як однотипні функції, тобто функції з однаковим списком параметрів і однаковим типом значення, яке вони повертають. Адреси виконавчих функцій присвоюють елементам масиву вказівників. У процесі роботи програми кожен вибір пункту меню чи перехід до іншого виконавчого блоку перетворюються у номер відповідної функції в масиві вказівників. Відбувається звертання до вибраної функції за її адресою і виконуються дії, встановлені для цієї функції. Така організація вибору потрібної функції вкрай проста – фактично це вибір елемента з масиву вказівників – і дає змогу уникнути громіздких розгалужень, які довелось би реалізовувати через багаторазові звертання до умовного оператора чи через використання оператора `switch`.

Проілюструємо застосування масиву вказівників прикладом програми, в якій користувач вибирає дії зі запропонованих у простому меню. Щоб зосередити увагу на механізмі вибору функцій, які реалізують пункти меню, в тілі кожної функції меню виконується тільки виведення інформаційних повідомлень, а інші дії закоментовано. Робота програми закінчується, коли виконано пункт меню "Завершення роботи". Особливості організації окремих функцій роз'яснено в коментарях, а після тексту програми наведено додаткові пояснення.

```
/*********************************************/
/*  Приклад організації користувацького меню  */
/*********************************************/
#include <stdio.h>
#include <conio.h>
typedef void  ACTION (void * pred);          /* тип функцій меню */
ACTION Item1;                                /* прототипи функцій */
ACTION Item2;
ACTION Item3;
ACTION ExitItem;
void Menu (void);
ACTION * GetAction (void);
```

```
int main (void)
{
    ACTION *pact,                    /* вказівники на функції вибраного і */
           *pdone = NULL;            /* попередньо виконаного пункту меню */
  Menu();                                          /* висвітлення меню */
  do {                                        /* цикл виконання пунктів меню */
     pact = GetAction();                       /* вибір функції пункту меню */
     (*pact)(pdone);                          /* виконання вибраного пункту */
     pdone = pact;                          /* фіксація виконаного пункту меню */
  } while (pdone != ExitItem);             /* поки не виконано пункт виходу */
  return 0;
}
void Menu (void)              /* функція формування меню */
{
  clrscr();                                              /* очищення екрана */
  puts ("\t   Набір дій:");
  puts ("\t 1 - Варіант дій 1");
  puts ("\t 2 - Варіант дій 2");
  puts ("\t 3 - Варіант дій 3");
  puts ("\t 4 - Завершення роботи");
}
ACTION* GetAction (void)  /* функція вибору функції пункту меню */
{
    ACTION* fun_arr[] = {Item1, Item2, Item3, ExitItem};        /* масив
                                  вказівників на функції пунктів меню */
    int num, nitems;
  nitems = sizeof(fun_arr) / sizeof(ACTION*);      /* кількість пунктів */
  do {                                        /* введення номера пункту меню */
     printf ("\nВибір -> ");
     num = getche()-'0';                    /* перетворення символа в число */
     printf("\n");
  } while (num < 1 || num > nitems);        /* повторення у разі помилки */
  return fun_arr[num-1];  /* повернення вказівника на вибрану функцію */
}
ACTION Item1                  /* функція пункту 1 меню */
{
  if (pred == NULL) {
     puts("    ** Початок дій **");
     /* . . .  - дії п.1, коли розпочинається робота програми */
  }
  /* . . .  - інші дії п.1. */
  puts ("   Виконано завдання п.1.");
}
ACTION Item2                  /* функція пункту 2 меню */
{
  /*  . . .  - дії п.2. */
  puts ("   Виконано завдання п.2.");
}
```

```
ACTION Item3                    /* функція пункту 3 меню */
{
   /*  . . .  - дії п.3 */
   if ((ACTION*)pred == Item2) {       /* якщо попереднім був пункт 2 */
      puts("   Виконано завдання п.3 (після п.2).");
      return;
   }
   /*  . . .  - інші дії п.3 */
   puts ("   Виконано повне завдання п.3.");
}
ACTION ExitItem                 /* функція пункту 4 меню */
{
   /*  . . .   - завершальні дії програми */
   puts("    ** Роботу програми завершено **");
}
```

Приклад виконання:

```
        Набір дій:
     1 - Варіант дій 1
     2 - Варіант дій 2
     3 - Варіант дій 3
     4 - Завершення роботи
Вибір -> 1
    ** Початок дій **
   Виконано завдання п.1.
Вибір -> 3
   Виконано повне завдання п.3.
Вибір -> 2
   Виконано завдання п.2.
Вибір -> 3
   Виконано завдання п.3 (після п.2).
Вибір -> 4
    ** Роботу програми завершено **
```

Розділи меню реалізовано в програмі як набір однотипних функцій: `Item1()`, `Item2()`, `Item3()`, `ExitItem()`, які не повертають значення та мають один формальний параметр – вказівник `pred` з базовим типом `void`. Тип цих функцій задекларовано як тип `ACTION`. Параметр функцій використовується в `main()` для передавання адреси тієї функції, що виконувалась перед цим, у функцію, яка викликається для реалізації вибраного пункту меню. Функція `Item1()` аналізує отримане значення, щоб перевірити, чи це перше звертання до меню, а функція `Item3()` реалізує різні дії залежно від того, який пункт меню їй передував (зверніть увагу на операцію перетворення типу `(ACTION*)pred`, що необхідна для порівняння функцій).

Функція `main()` керує циклічним процесом роботи з меню. Першою викликається функція `GetAction()`, яка повертає адресу тієї функції, що повинна реалізовувати вибраний користувачем пункт меню. Ця адреса записується у вказівник `pact`.

Наступний оператор

```
(* pact)(pdone);
```

викликає функцію за адресою, яка записана в `pact`, і передає їй через параметр `pdone` адресу функції, що виконувалась перед цим. Коли робота функції `(*pact)()` завершується, `pdone` отримує її адресу, і процес звертання до меню повторюється знову. Робота програми припиняється, коли виконано завершальну функцію `ExitItem()`.

Вибір функції, що відповідає вибраному користувачем пункту меню, виконує функція `GetAction()`. Функція працює з масивом вказівників на функції `fun_arr`, елементи якого проініціалізовані адресами відповідних функцій пунктів меню:

```
ACTION * fun_arr[] = { Item1, Item2, Item3, ExitItem };
```

Кількість елементів масиву визначається кількістю імен функцій у списку ініціалізації:

```
nitems = sizeof(fun_arr)/sizeof(ACTION*);
```

У разі виклику `GetAction()` відбувається зчитування клавіші з номером вибраного пункту меню і формується число `num=getche()-'0'`. Якщо `num` потрапляє в діапазон `1..nitems`, то в функцію `main()` передається адреса функції вибраного пункту меню, інакше процес вибору повторюється. Адреса функції вибраного пункту меню задається як елемент масиву вказівників `fun_arr` з індексом `num-1`.

11.10. Рекурсивні функції

Рекурсивними називаються функції, які в процесі виконання звертаються самі до себе. Рекурсія може бути безпосередньою (*пряма рекурсія*) – коли з тіла функції викликається та ж сама функція, або посередньою (*непряма рекурсія*) – коли функція викликає інші функції, які в свою чергу безпосередньо або через треті функції звертаються до даної функції.

Здебільшого рекурсивні функції є лаконічними в записі й дають змогу наочно та стисло відобразити алгоритм розв'язування задачі. Існує цілий ряд задач, яким рекурсивні алгоритми й конструкції найбільш притаманні, тому запрограмувати такі задачі без використання рекурсії достатньо складно, а деколи практично неможливо. З іншого боку, рекурсивні звертання часто не легкі для сприйняття, аналізу та контролю. Вони можуть вимагати значних обсягів оперативної пам'яті та призводити до часових затримок у процесі реалізації програми. Основні засади організації рекурсивних функцій, переваги та недоліки їх застосування для різних задач пояснимо на прикладах кількох простих функцій, що використовують безпосередню рекурсію.

Рекурсивне та ітераційне обчислення ряду Фібоначчі. Першою розглянемо функцію, призначену для знаходження n-го числа Фібоначчі. За математичним означенням два перших числа Фібоначчі (з номерами 1 та 2) дорівнюють 1, а кожне наступне обчислюється як сума двох попередніх. Означення ряду Фібоначчі є рекурентним (щоб знайти n-е число, треба просумувати $(n-1)$-е та $(n-2)$-е числа Фібоначчі), тому для його програмної реалізації спробуємо застосувати відповідну рекурсивну функцію.

```
/*  Визначення числа Фібоначчі за заданим номером  */
long  FibonNumb (int n)
{
   if (n == 1 || n == 2)  return 1;              /* перше та друге число */
   return FibonNumb(n-1) + FibonNumb(n-2);        /* всі наступні числа */
}
```

Наведена функція програмно відтворює означення числа Фібоначчі. Вона дуже коротка (перевірку допустимості значення параметра n опущено навмисно), тому наочно демонструє, як організовані рекурсивні звертання в функціях.

За способом реалізації рекурсивні виклики функцій нічим не відрізняються від звичайних викликів: кожен раз створюється новий набір формальних параметрів і внутрішніх змінних функції, а код функції виконується від самого початку. Результат виконання функції повертається в точку її виклику.

Щоб детальніше пояснити, як виконуються рекурсивні виклики, звернемось до функції `FibonNumb()` для знаходження, наприклад, 12-го за порядковим номером числа Фібоначчі. Оскільки 12>2, то з двох операторів тіла функції `FibonNumb()` вибирається другий, який фактично буде таким:

```
return  FibonNumb(11) + FibonNumb(10);
```

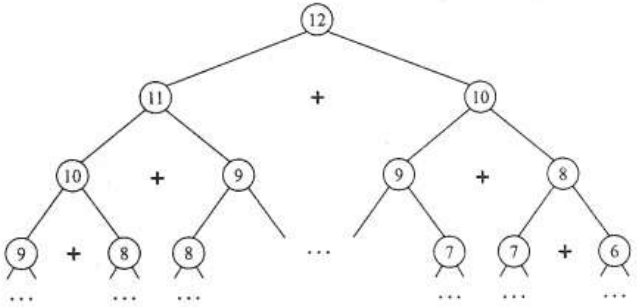


Рис. 11.2. Схема рекурсивних викликів FibonNumb() для обчислення 12-го числа Фібоначчі

Щоб реалізувати цей оператор, необхідно двічі викликати функцію `FibonNumb()` для менших за значенням аргументів. У свою чергу, обчислення `FibonNumb(11)` потребує обчислення 10-го і 9-го чисел Фібоначчі, а для знаходження 10-го числа треба викликати `FibonNumb(9)` та `FibonNumb(8)` і т. д. (рис. 11.2). Цей процес називають *рекурсивним зануренням*. Виклик `FibonNumb(3)` зумовить виконання оператора

```
return  FibonNumb(2) + FibonNumb(1);
```

Обидва звертання цього оператора повернуть значення 1, оскільки це значення двох перших чисел Фібоначчі, які становлять базис функції. Процес занурення по даній вітці припиняється і розпочинається зворотний процес *рекурсивного повернення*. Сума `FibonNumb(1)` та `FibonNumb(2)` є значенням, яке повертає функція `FibonNumb(3)`.

Обчислене значення передається у функцію, яка реалізує виклик `FibonNumb(4)`. Але, щоб знайти четверте число Фібоначчі, ця функція повинна також реалізувати повторне звертання до `FibonNumb(2)`. Значення `FibonNumb(4)` передається у функцію, яка обчислює п'яте число Фібоначчі і т. д. Процес рекурсивного повернення завершиться, коли повністю виконається найперший виклик функції, тобто `FibonNumb(12)`.

Обов'язковим елементом кожної рекурсивної функції повинна бути *умова завершення* роботи. В функції `FibonNumb()` вона задається оператором

```
if (n == 1 || n == 2)  return 1;
```

який становить *базис* даної рекурсії. У разі відсутності умови завершення або її помилковості рекурсивні виклики призведуть до "зависання" програми, оскільки рекурсивна функція буде викликати сама себе безконечно.

Реалізація наведеної функції, яка вимагає великої кількості повторних викликів `FibonNumb()` для тих самих значень аргументів, вочевидь неефективна (пропонуємо самим підрахувати, скільки раз буде викликатись, наприклад, `FibonNumb(4)` у процесі обчислення 12-го числа Фібоначчі). Для порівняння запишемо ітераційний варіант функції обчислення заданого числа Фібоначчі.

```
/*  Визначення числа Фібоначчі - ітераційний варіант  */
long  FibonNumb_iter (int n)
{
     int k;
     long fibp1, fibp2, fib;
   if (n < 3) return 1;
   fibp1 = fibp2 = 1;          /* два визначені числа */
   k = 2;                      /* номер останнього визначеного числа */
   do {
      fib = fibp1+fibp2;       /* наступне число */
      fibp2 = fibp1;           /* зміна значень попередніх чисел */
      fibp1 = fib;
   } while (++k != n);
   return fib;
}
```

За простотою та наочністю ця функція поступається рекурсивній. Проте в процесі роботи вона одноразово обчислює значення кожного числа Фібоначчі, внаслідок чого її швидкодія значно вища, ніж у попередньої рекурсивної функції. Щоб відчути різницю в часі виконання цих двох функцій, спробуйте знайти 45-те число Фібоначчі (передостаннє з чисел, що потрапляють у діапазон даних з типом `long int`).

Рекурсія в алгоритмі Евкліда. Наведемо ще один приклад рекурсивної функції. Вона призначена для знаходження найбільшого спільного дільника (НСД) двох цілих чисел за відомим рекурсивним методом Евкліда. Цей метод базується на тому, що НСД двох цілих чисел збігається з НСД меншого з цих чисел і залишку від ділення більшого числа на менше.

```
/*  Визначення НСД двох цілих чисел  */
unsigned NSD (unsigned a, unsigned b)
{
   if (a % b == 0)  return b;          /* b - НСД */
   if (a < b)  return NSD (b, a);      /* переставлення чисел місцями */
   return NSD (b, a % b);              /* рекурсивне зменшення чисел  */
}
```

Дана функція не тільки лаконічна в записі та проста для читання і розуміння. Вона також ефективна й достатньо швидка, оскільки рекурсивні виклики здійснюються лінійно, а глибина їх вкладень (це властивість алгоритму Евкліда) не буває великою. Наприклад, обчислення `NSD(405, 1026)` зумовить виклик `NSD(1026, 405)`, а далі послідовно викликатимуться `NSD(405, 216)`, `NSD(216, 189)` і `NSD(189, 27)`. Останній виклик функції поверне значення 27, яке буде передане у функцію, що здійснила цей виклик. У свою чергу, ця функція відразу поверне отримане значення в попередню функцію, оскільки виконується оператор

```
return NSD(b, a % b);
```

Зворотний процес завершиться після повернення в функцію, яка зробила найперший виклик: `NSD(405, 1026)`.

Хвостова та зворотна рекурсія. Рекурсивними викликами можна замінити циклічні процеси. Подамо приклад короткої функції, призначеної для виведення на екран `k` послідовних елементів заданого масиву дійсних чисел. Функція використовує рекурсивне звертання замість традиційного для задач такого роду оператора циклу.

```
/*  Виведення масиву дійсних чисел  */
void PrintArray (double arr[], int k)
{
   if (k == 0)  return;                /* елементів немає */
   printf ("%8.2lf", *arr);            /* виведення першого елемента   */
   PrintArray(arr+1, k-1);             /* продовження для решти масиву */
}
```

У всіх трьох розглянутих вище функціях рекурсивні звертання були останніми операторами функцій. Такий варіант рекурсії називають *хвостовим*. Проте інколи доцільно застосувати іншу організацію рекурсивної функції. Розглянемо наступну функцію, яка подібна до `PrintArray()`, але внаслідок зміни порядку операторів роздруковує масив у зворотній послідовності – від останнього елемента до першого.

```
/*  Виведення масиву дійсних чисел у зворотному порядку  */
void ReversPrint (double arr[], int k)
{
   if (k > 1)
      ReversPrint(arr+1, k-1);         /* просування по масиву */
   printf ("%8.2lf", *arr);            /* виведення елементів  */
}
```

Виведення елементів масиву розпочнеться тільки тоді, коли `k` дорівнюватиме `1`, отже `arr` вказуватиме на останній елемент масиву. Буде надруковано цей елемент і керування перейде у викликаючу функцію. У цій функції ще залишився не виконаним останній оператор, що друкує `*arr`, тому на екран виводиться значення передостаннього елемента масиву і продовжується зворотний процес рекурсивного повернення.

Два останні приклади демонструють важливі властивості рекурсивних функцій:

1) оператори, записані перед рекурсивним викликом, виконуються в тому ж порядку, в якому відбуваються виклики функції;
2) оператори, розташовані після рекурсивного виклику, виконуються в зворотному порядку відносно рекурсивних викликів даної функції.

Кожен виклик рекурсивної функції пов'язаний зі створенням окремого набору параметрів і внутрішніх змінних – саме їх значення використовуються у процесі реалізації даного виклику. Для збереження даних, що створюються у процесі звертання до функції, використовується спеціальна область оперативної пам'яті, яка називається *стеком*. Особливість організації стека в тому, що записані в нього дані зчитуються в послідовності, зворотній до послідовності їх запису (таку форму запису/читання даних ще називають *LIFO – last in, first out* – останнім прийшов, першим вийшов). З кожним звертанням до функцій програми стек збільшується внаслідок запису параметрів і внутрішніх змінних цієї функції. Дані активізованої функції зберігаються у стеку до моменту завершення її роботи, після чого пам'ять, зайнята даними функції, звільняється. Найпершою може завершити свою роботу тільки функція, що була викликана останньою. Тому послідовність звільнення стека є зворотною до порядку виклику функцій у програмі. Якщо реалізація певної задачі пов'язана зі значною кількістю рекурсивних звертань до функції (прикладом може слугувати функція `FibonNumb()` у разі великих значень параметра `n`), то цілком імовірною стає загроза переповнення стека. Фізичний обсяг стека залежить від апаратно-програмних особливостей комп'ютера і встановленої моделі пам'яті.

Рекурсивний пошук та інші задачі. Останній приклад – рекурсивна функція для пошуку заданого елемента у масиві цілих чисел, впорядкованих за зростанням значень. Функція повертає вказівник на знайдений елемент (у разі потреби індекс елемента можна визначити через різницю адрес знайденого елемента і початку масиву) або `NULL`, якщо такого елемента немає. З урахуванням впорядкованості елементів масиву, застосуємо для пошуку метод половинного ділення. Перевіривши серединний елемент, визначаємо, в якій половині масиву: молодшій чи старшій – має бути розташоване шукане значення. Далі аналогічним чином перевіряємо тільки цю частину масиву. Пошук завершується, коли елемент знайдено, або коли перевірено останній можливий елемент масиву.

Порівняно з лінійним пошуком, який вимагає послідовного перегляду елементів масиву (можливо, що всіх), пошук за методом половинного ділення дає змогу значно зменшити кількість операцій перевірки. У найгіршому випадку лінійний пошук вимагатиме N операцій порівняння елементів, а бінарний – тільки $[\log_2 N]+1$, де N – кількість елементів масиву, а квадратні дужки [] позначають цілу частину числа.

```
/*  Пошук заданого елемента nfind у впорядкованому масиві  */
int*  FindElement (int nfind, int arr[], int k)
{
     int* pel=arr+k/2;      /* вказівник на серединний елемент масиву */
   if (*pel==nfind) return pel;                    /* число знайдено */
   if (k==1) return NULL;           /* масив перевірено - числа немає */
   if (*pel>nfind)     /* пошук продовжується в лівій половині масиву */
      return  FindElement(nfind, arr, k/2);
   else                  /* пошук продовжується у правій половині масиву */
      return  FindElement(nfind, pel+1, k-k/2-1);
}
```

Звісно, метод половинного ділення не складно відтворити ітераційно. Проте, оскільки глибина рекурсії наведеної функції не перевищує $[\log_2 N]$, то застосування простої у записі та наочної рекурсивної функції цілком виправдане.

Рекурсивний алгоритм швидкого сортування Хоара, який належить до найбільш ефективних методів сортування, теж пов'язаний із поділом масиву на частини, кожна з яких сортується окремо за тим самим принципом [12, 24]. Програмна реалізація цього алгоритму ітераційним методом набагато складніша.

Щоб більше дізнатись про рекурсію та алгоритми, для яких вона доцільна або взагалі невід'ємно притаманна, рекомендуємо звернутись до літератури [4, 12, 14, 23]. Зауважимо тільки, що рекурсивні алгоритми та підходи широко використовуються для роботи з динамічними структурами даних, передусім з інформаційними деревами, для розв'язування комбінаторних задач, для програмування задач із графами, а також цілої низки задач, що належать до класу задач штучного інтелекту.

11.11. Функції з неоголошеними параметрами

Мова С дає змогу створювати функції, в оголошенні яких вказуються не всі формальні параметри, а тільки один або декілька початкових. Такі функції називають також функціями *зі змінною кількістю* (або *змінним складом*) *параметрів*.

Синтаксис оголошення функцій зі змінною кількістю параметрів такий:

тип_значення_функції ім'я_функції (*оголошення обов'язкових_параметрів*, ...);

У списку параметрів функції треба обов'язково оголошувати один або декілька початкових параметрів (їх називають *обов'язковими*), після яких записують трикрапку ..., якою позначають можливу наявність наступних неоголошених параметрів функції (їх називають *необов'язковими*).

У викликах функцій зі змінною кількістю параметрів спочатку треба вказати значення всіх обов'язкових параметрів, а після них можна записати довільну кількість даних, що задають неоголошені параметри функції. Компілятор контролює тільки обов'язкові фактичні параметри, типи яких відомі, а кількість і типи неоголошених параметрів компілятор не перевіряє.

Якщо прототип функції оголошено так:

```
char * OutList(char * list, ...);
```

то `OutList()` розглядатиметься як функція зі змінною кількістю параметрів, що повертає вказівник на символьний рядок. У звертаннях до цієї функції, крім обов'язкового першого параметра, що повинен мати тип `char *`, можна вказувати довільну кількість інших значень. Наприклад, звертання може бути таким:

```
pnext = OutList("3s", str1, str2, str3);
```

Найбільш відомими бібліотечними функціями зі змінною кількістю та складом параметрів є функції `printf()` та `scanf()`, список формальних параметрів яких оголошено подібно до наведеного вище в оголошенні функції `OutList()`.

Реалізація функцій з неоголошеними параметрами базується на механізмі взаємодії фактичних і формальних параметрів. Нагадаємо, що в разі виклику функції послідовно обчислюються і заносяться у стек значення її фактичних параметрів. Якщо функцію оголошено з повним списком параметрів, то перед записом у стек компілятор перевіряє відповідність типу кожного фактичного параметра і в разі потреби перетворює його значення до типу відповідного формального параметра. Для функцій зі змінним складом параметрів контроль типів здійснюється тільки для обов'язкових параметрів, а значення неоголошених параметрів не перевіряються. У стеку значення всіх параметрів функції розташовуються підряд. Кожне значення займає ділянку пам'яті, відповідну до розміру його типу (додатково перед записом у стек дані з типом `char` перетворюються в `int`, а дані з типом `float` – у тип `double`). Оскільки функції зі змінним складом параметрів повинні мати хоча б один обов'язковий параметр, то можна визначити адресу першого неоголошеного параметра – він буде розташованим за останнім обов'язковим параметром, тип якого, а отже й розмір, відомі. Щоб звертатись до наступних неоголошених параметрів, треба певним чином передати в функцію їх типи і кількість, що дасть змогу визначити адресу кожного з неоголошених параметрів.

11.11.1. Безпосереднє звертання до неоголошених параметрів

Перші дві функції з неоголошеними параметрами, які розглянемо далі, базуються на схемі запису в стек значень фактичних параметрів, що властива, зокрема, системі програмування Borland C. Першим у стек заноситься значення останнього фактичного параметра, тому його адреса буде найбільшою (стек заповнюється від вершини). Далі послідовно заносяться значення всіх інших фактичних параметрів, адреса кожного з них буде меншою на розмір типу, який має значення цього параметра. Найменшу адресу в оперативній пам'яті матиме значення першого параметра функції.

Використаємо властивість збереження параметрів у стеку за порядком зростання їх адрес, щоб створити функцію, яка визначає середнє квадратичне значення (СКЗ) довільної кількості дійсних аргументів. В оголошенні функції вказуються два обов'яз-

ковых параметри: k – кількість чисел, що опрацьовуються у поточному звертанні до функції, та x – значення першого аргументу.

```
/*****************************************************************/
/* Обчислення середньоквадратичного значення набору дійсних чисел */
/* за допомогою функції зі змінною кількістю параметрів          */
/*****************************************************************/

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

double SqVal1 (unsigned k, double x, ... )
{
     double *p, sum=0.0;
   for (p=&x;  k>0;  k--, p++)      /* цикл сумування квадратів чисел */
      sum += *p**p;                       /* тобто sum=sum+(*p)*(*p); */
   return sqrt(sum);
}

void main(void)
{
     double sqv, a=11.53, b=2.46;
   sqv=SqVal1(3, 2.8, 4.0, 6.9);              /* СКЗ трьох значень     */
   printf("Тест 1:  СКЗ = %5.2lf\n", sqv);
   sqv=SqVal1(4, a, b, a*b, a/b);             /* СКЗ чотирьох значень */
   printf("Тест 2:  СКЗ = %5.2lf\n", sqv);
}
```

Результат виконання:

```
Тест 1:  СКЗ =  8.45
Тест 2:  СКЗ = 31.07
```

Для просування по неоголошених дійсних аргументах у функції `SqVal1()` використано вказівник `p`, який початково отримує адресу першого дійсного числа. Вираз `p++` просуває цей вказівник на `sizeof(double)` байт, тобто встановлює його на початок наступного параметра функції.

Оскільки типи неоголошених параметрів не контролюються і не перетворюються, то у викликах даної функції всі значення, для яких обчислюється СКЗ, повинні мати тип `double` (треба обов'язково записувати `4.0`, а не `4`).

Кількість чисел, які треба опрацювати в поточному виклику функції `SqVal1()`, передається в цю функцію через її перший параметр. Іншим поширеним способом фіксування кількості необов'язкових параметрів є використання деякого маркера (ознаки) кінця списку аргументів. Наступний варіант функції обчислення СКЗ `SqVal2()` спирається на припущення, що всі дані, СКЗ яких обчислюється, будуть додатними, тому роль маркера кінця списку параметрів виконує довільне від'ємне число.

```
/*****************************************************************/
/*  Обчислення СКЗ. Варіант 2 функції з неоголошеними параметрами  */
/*****************************************************************/
#include <stdio.h>
#include <math.h>
double  SqVal2 (double x, ... )
{
     double sum=x*x, *p=&x;
   while (*++p > 0)          /* розглядаються тільки додатні аргументи */
      sum += *p**p;                        /* сумування квадратів чисел */
   return sqrt(sum);
}
void main(void)
{
     double sqv, a = 11.53, b=2.46;
   sqv = SqVal2(3, 2.8, 4.0, 6.9, -1.0);      /* СКЗ трьох значень    */
   printf("Тест 3:  СКЗ = %5.2lf\n", sqv);
   sqv = SqVal2(4, a, b, a*b, a/b, -0.5);     /* СКЗ чотирьох значень */
   printf("Тест 4:  СКЗ = %5.2lf\n", sqv);
}
```

Результат виконання:

```
Тест 3:  СКЗ =  8.45
Тест 4:  СКЗ = 31.07
```

В обох розглянутих функціях всі неоголошені параметри мали єдиний встановлений тип. Проте часто треба забезпечити опрацювання неоголошених параметрів з різними типами. У цьому випадку необхідно якимось способом передати у функцію тип кожного параметра, щоб забезпечити можливість коректно визначати адреси параметрів у стеку. Функції `printf()` та `scanf()` для цієї мети використовують обов'язковий параметр – рядок формату, специфікації якого визначають кількість і типи всіх наступних параметрів. У третьому варіанті функції обчислення СКЗ `SqVal3()` (її текст наведено в наступному параграфі) використано подібний прийом: першим серед параметрів, що передаються у функцію, вказується символьний рядок, кожна літера якого задає тип відповідного неоголошеного аргументу.

11.11.2. Макрозасоби для роботи з неоголошеними параметрами

У процесі визначення адрес неоголошених параметрів у функціях `SqVal1()` та `SqVal2()` використовувалась властивість послідовного розміщення параметрів у стеку за порядком зростання адрес. Але така схема збереження параметрів функції, коли останнім заноситься у стек перший параметр, не обов'язкова. Тому в системах програмування, орієнтованих на іншу організацію збереження даних, ці функції будуть хибними.

Бібліотека С містить спеціальні функції-макроси, задекларовані в заголовному файлі <stdarg.h>, які забезпечують простий доступ до неоголошених параметрів функції, незалежний від конкретної програмно-апаратної організації збереження параметрів. Використання стандартних функцій-макросів гарантує мобільність користувацьких функцій зі змінним складом параметрів.

Основних функцій-макросів, визначених у <stdarg.h>, три:

```
void va_start (va_list parg, останній_параметр);
тип va_arg (va_list parg, тип);
void va_end (va_list parg);
```

Порядок роботи з цими макросами наступний. У функції зі змінним складом параметрів треба оголосити спеціальний вказівник з типом va_list, який надалі буде використовуватись для звертання до необов'язкових параметрів. Цей вказівник встановлюється на перший неоголошений параметр через звертання до функції-макроса:

```
va_start (вказівник, останній_обов'язковий_параметр);
```

тут другим аргументом вказується ім'я останнього з обов'язкових параметрів. У процесі опрацювання неоголошених параметрів використовують функцію-макрос

```
тип_параметра va_arg (вказівник, тип);
```

що повертає значення неоголошеного параметра, адресу якого в даний момент зберігає *вказівник*, який має тип va_list. Другий аргумент va_arg() – *тип* задає тип поточного неоголошеного параметра. Він має бути або стандартним ключовим словом: int, double, char * тощо, або попередньо задекларованим користувацьким найменуванням типу. Після кожного звертання до неоголошеного параметра *вказівник* автоматично пересувається на наступний параметр. Як і в попередніх прикладах, треба встановити певний спосіб обмеження кількості параметрів. Коли всі параметри опрацьовано, перед поверненням із функції викликають макрос

```
va_end (вказівник);
```

який необхідний для коректного завершення роботи з неоголошеними параметрами.

Проілюструємо використання описаних функцій-макросів прикладом третього варіанта функції обчислення СКЗ, в якому передбачено, що неоголошені параметри можуть мати різні типи. Тип кожного аргументу задається відповідною літерою у рядку типізації: i – int, l – long, d – double. Рядок символів типізації types – єдиний обов'язковий параметр функції SqVal3().

```
/*****************************************************************/
/* Обчислення СКЗ. Варіант 3 функції з неоголошеними параметрами - */
/* використання стандартних макросів <stdarg.h>                   */
/*****************************************************************/
#include <stdio.h>
#include <math.h>
#include <stdarg.h>
```

```
double  SqVal3 (char *types, ... )
{
     double sum=0, dn;
     int in;  long ln;
     char *pt=types;
     va_list parg;      /* вказівник на список неоголошених параметрів */
   va_start (parg, types);            /* початок роботи з параметрами */
   while (*pt!='\0') {                 /* цикл по рядку специфікацій */
      switch (* pt) {                   /* визначення типу параметра */
        case 'i':  in=va_arg(parg, int);
           sum+=(double)in*in;   break;
        case 'l':  ln=va_arg(parg, long);
           sum+=(double)ln*ln;   break;
        case 'd':  dn=va_arg(parg, double);
           sum+=dn*dn;
      }
      pt++;
   }
   va_end (parg);                    /* завершення роботи з параметрами */
   return sqrt(sum);
}
int main(void)
{
     double sqv, a=11.53;
     int k = 5;
     long g = 7;
   sqv = SqVal3("iidi", 8, 4, 6.9, 11);       /* СКЗ чотирьох значень */
   printf("Тест 5:  СКЗ = %5.2lf\n", sqv);
   sqv = SqVal3("dlild", a, g, k, 2*g, a/k);     /* СКЗ п'яти значень */
   printf("Тест 6:  СКЗ = %5.2lf\n", sqv);
   return 0;
}
```

Результат виконання:

```
Тест 5:  СКЗ = 15.77
Тест 6:  СКЗ = 20.21
```

Запитання та завдання для самоконтролю

1. Яке призначення функцій у С-програмах?
2. Яка структура заголовка функції? Як оголошуються формальні параметри функцій? Яке значення може повертати функція?
3. Що таке прототип функції? У яких випадках застосовують прототипи функцій? Де розміщують прототипи функцій в програмі?

4. Знайдіть п'ять помилок у наступному описі функції:

```
void  ChgFst (double a, b; int k);
{
   a = (a>b) ? b/k : b*k;
   return a
}
```

5. Як звертаються до функцій? Чи треба узгоджувати фактичні параметри функції з формальними? Який механізм їх взаємодії?
6. Як можна повернути з функції декілька різних значень?
7. Як оголошуються формальні параметри функції, через які буде здійснюватись звертання до масивів або символьних рядків?
8. Функція повинна формувати новий символьний рядок. Як передати цей рядок у викликаючу функцію? Яке значення найчастіше повертають такі функції?
9. Які дані називають параметрами командного рядка? Яким чином можна звертатись до них у С-програмах?
10. У програмі треба розробити функцію, яка буде опрацьовувати почергово дві матриці, оголошені так:

```
float matr1[25][NC], matr2[40][NC];
```

Які з поданих нижче оголошень відповідного формального параметра функції опрацювання матриць правильні?

1) `float m[][]`
2) `float *m[NC]`
3) `float (*m)[NC]`
4) `float (*m)[]`
5) `float **m`
6) `float m[][NC]`

11. Чи можуть структури бути параметрами функції? Чи можна повертати із функції цілу структуру? Які способи опрацювання структур у функціях найбільш ефективні?
12. Як оголошується вказівник на функцію? Які значення можна йому присвоювати? Де застосовують вказівники на функцію?
13. Прототип функції `Test()` у програмі оголошено так:

```
int  Test (int k, int (*q)(char *));
```

Що може бути другим параметром у викликах даної функції?

14. Як зміниться прототип функції `Test()` із п.13, якщо в програмі перед ним ввести декларацію:

```
typedef  int  Tfun (char *);
```

15. Знаючи декларацію `Tfun`, прочитайте та поясніть наступні оголошення:

```
1) Tfun  *f1, *f2, newfn;
2) Tfun*  fset[10];
3) Tfun*  ResFun (int n);
4) int  ContrDat (Tfun* func1, Tfun* func2);
```

16. Які функції називають рекурсивними? У чому переваги і недоліки рекурсивних функцій? Що таке хвостова та зворотна рекурсія, чим відрізняється результат застосування цих двох видів рекурсії?

17. Як оголошуються функції зі змінним складом параметрів? Як визначають реальну кількість і типи параметрів у реалізаціях таких функцій? Як звертаються до неоголошених параметрів у тілі функції?

18. Яке призначення макрозасобів заголовного файла `<stdarg.h>`?

Для наведених нижче задач запрограмуйте відповідні функції та перевірте їх правильність на різних тестових наборах фактичних параметрів

19. Розробити функцію, параметрами якої є два цілих числа. Функція повинна повертати частку від ділення більшого з цих чисел на менше.

20. Розробити функцію, яка перевіряє, чи заданий рік є високосним. *Підказка*: високосними (що мають 29 днів у лютому) вважаються роки, які цілочислово діляться на 4, за винятком тих, що діляться на 100, окрім кратних до 400 (1900 р. – невисокосний, 2000 р. – високосний).

21. Розробити функцію, яка змінює значення двох заданих дійсних змінних. Перша змінна повинна отримати значення суми початкових даних, а друга – значення їх різниці.

22. Розробити функцію, яка видаляє зі заданого символьного рядка всі цифрові символи і повертає кількість видалених цифр.

23. Розробити функцію, яка обчислює скалярний добуток двох векторів дійсних чисел, що мають однакову розмірність.

24. Розробити функцію, яка виділяє зі заданого речення і виводить на екран усі слова, в яких зустрічається подвоєння літери (наприклад, Ілля).

25. Розробити функцію, яка формує символьний рядок, що відповідає двійковому коду заданого довгого цілого беззнакового числа.

26. Розробити функцію, яка формує квадратну матрицю розмірністю N×N, заповнену послідовністю натуральних чисел: $1, 2, 3, \ldots N^2$, розташованих у формі спіралі. Приклад для N=5:

```
 1  2  3  4  5
16 17 18 19  6
15 24 25 20  7
14 23 22 21  8
13 12 11 10  9
```

27. Розробити функцію, яка сортує масив дійсних чисел за порядком зростання значень елементів. Використовуючи розроблену функцію, впорядкувати кожен рядок заданої прямокутної матриці.

28. Розробити рекурсивний варіант функції піднесення заданого дійсного аргументу до цілого степеня.

29. Доповнити функцію з п. 28 так, щоб вона реалізувала т. зв."індійський метод" піднесення до степеня:

$$x^n = \begin{cases} 1, & \text{якщо } n = 0; \\ (x^{n/2})^2, & \text{якщо } n \text{ парне}; \\ x(x^{n/2})^2, & \text{якщо } n \text{ непарне}. \end{cases}$$

30. Розробити функцію з неоголошеними параметрами, яка визначає і повертає значення найменшого зі своїх цілочислових аргументів, кількість яких вказується у першому параметрі функції.

Розділ 12

КЛАСИ ПАМ'ЯТІ ДАНИХ

У цьому розділі:

- Поняття класу пам'яті, часу існування та області видимості об'єктів програми
- Класифікація об'єктів за часом існування та областю видимості
- Глобальні та локальні змінні, область їх дії, автоматична ініціалізація
- Вплив специфікаторів `auto`, `register`, `static` та `extern` на час існування та область видимості локальних змінних
- Специфікатори класів пам'яті глобальних змінних
- Область видимості об'єктів у разі багатофайлових програм
- Створення проектів програм

12.1. Клас пам'яті, час існування та видимість об'єкта

Кожен об'єкт програми (змінна, константа, функція) має свій тип і клас пам'яті. Як вже зазначалось, *тип* визначає обсяг пам'яті, який виділяється для даного об'єкта, і операції, що можуть виконуватись над об'єктом. *Клас пам'яті* задає місце розташування об'єкта в оперативній пам'яті та встановлює для нього *час існування*, тобто час, протягом якого об'єкт зберігається в оперативній пам'яті, і *область видимості*, яка визначає ту частину програми, де можна використовувати цей об'єкт. На відміну від типу, який у С-програмах завжди треба задавати явно, клас пам'яті можна явно не вказувати, тоді він встановлюється компілятором автоматично за місцем оголошення об'єкта в програмі. В усіх наведених раніше програмах використовувалось автоматичне встановлення класу пам'яті.

Класифікація об'єктів за часом існування та областю видимості. За часом існування (часто говорять: *часом життя*) об'єкти поділяють на три групи:

- *глобальні* (або *статичні*) – ці об'єкти існують протягом усього часу виконання програми;
- *локальні* – пам'ять для таких об'єктів виділяється з входом у блок, де вони оголошені;

коли ж завершується виконання блоку, ці об'єкти стають невизначеними, а пам'ять, яку вони займали, вважається вільною;
- *динамічні* – пам'ять для таких об'єктів виділяється і звільняється за відповідною вимогою програми.

Глобальні та статичні дані зберігаються в окремій області оперативної пам'яті, яку в системах з сегментною організацією пам'яті називають *сегментом даних*; локальні дані зберігаються в області пам'яті, що називається *стеком*; а динамічні дані – в області пам'яті, що підлягає динамічному розподілу (особливості використання динамічної пам'яті для збереження даних програми розглянемо в наступних розділах).

Класи пам'яті об'єктів пов'язані з поняттям *програмного блоку*. Блоком у С вважається тіло функції, а також кожна внутрішня група описів та операторів у фігурних дужках. За правилами замовчування змінні, описані всередині блоку, та формальні параметри функцій мають локальний час існування, а змінні, описані зовні всіх блоків, тобто між функціями мають глобальний час існування. Відповідно говорять про *внутрішні* (локальні) та *зовнішні* (глобальні) оголошення об'єктів програми.

Функції в С-програмах можна описувати тільки на зовнішньому рівні, тому всі функції мають глобальний час існування.

За областю видимості (або *областю дії*) об'єкти також ділять на три групи:
- *глобальні* – видимі в межах усієї програми;
- *частково глобальні* – видимі в межах одного програмного файла;
- *локальні* – видимі в блоці, де оголошено даний об'єкт.

Область дії глобальних і локальних змінних. Переважно більшість змінних програми має локальний час існування та локальну видимість, а глобальними оголошують тільки окремі змінні, призначені для спільного використання. Діють такі правила:
- змінні, оголошені на зовнішньому рівні (між функціями), частково глобальні – областю їх дії є частина програми від точки оголошення до кінця файла;
- змінні, оголошені на внутрішньому рівні (всередині програмного блоку), а також формальні параметри функцій мають область дії від точки оголошення до кінця функції (блоку);
- змінні, що є параметрами прототипу функції, видимі тільки в межах цього прототипу;
- якщо ім'я внутрішньої змінної збігається з іменем якогось зовнішнього об'єкта, то в межах цього блоку внутрішня змінна "закриває" зовнішній об'єкт, а за межами блоку дія зовнішнього об'єкта відновлюється;
- клас пам'яті змінних можна встановити явно, використовуючи відповідні специфікатори класів пам'яті.

Всі глобальні та статичні змінні автоматично ініціалізуються нульовими значеннями, а для локальних змінних автоматична ініціалізація не виконується.

Проілюструємо сказане на прикладі змінних програми, яка формує масив із цілих випадкових трицифрових чисел, кожне з яких містить цифру 5. Пояснення наведемо після тексту програми.

```
/*****************************************************/
/*  Формування масиву випадкових трицифрових чисел,  */
/*  кожне з яких містить цифру 5                     */
/*****************************************************/
#include <stdio.h>
#include <stdlib.h>
#define MAX 30                          /* максимальна розмірність масиву */
unsigned a[MAX];                              /* масив випадкових чисел */
int n=10;                                  /* задана кількість елементів */
void CreateArray (void);                          /* прототипи функцій */
void PrintArray (void);
int Contains5 (unsigned n);
int main (void)
{
   CreateArray();                              /* створення масиву чисел */
   PrintArray();                             /* виведення створеного масиву */
   return 0;
}
void CreateArray (void)      /* функція формування масиву */
{
     int i=0;                          /* не ініціалізується автоматично */
   randomize();
   while (i<n) {                            /* звертання до глобального n */
     unsigned n;                              /* внутрішня змінна блоку */
     n=100+random(900);                /* генерування трицифрового числа */
     if (Contains5(n))                          /* у числі є цифра 5 */
          a[i++]=n;                           /* запис числа в масив */
   }
}
int Contains5 (unsigned a)  /* функція перевірки цифр числа */
{
   if (a%10==5 || a/10%10==5 || a/100==5)
      return 1;
   else
      return 0;
}
void PrintArray (void)       /* функція виведення масиву */
{
     int i;
   printf("\n\t\t Сформований масив:\n");
   for (i=0; i<n; i++)
      printf("%5d", a[i]);
}
```

Приклад виконання:

```
          Сформований масив:
 358  599  456  135  715  656  559  285  517  895
```

У програмі використано дві глобальні змінні: ім'я масиву `a` та кількість його елементів `n`. Ці змінні доступні в усіх функціях, перед якими вони описані, зокрема їх опрацьовують функції `CreateArray()` та `PrintArray()`.

З одного боку, обмін даними через глобальні змінні зручний, бо дає змогу зменшити кількість параметрів функцій і уникнути запису в стек копій фактичних аргументів. З іншого боку, застосування глобальних змінних робить функції менш універсальними і збільшує загрозу виникнення помилок, оскільки значення глобальних змінних можуть бути змінені з кожної точки програми. Тому доцільно глобальними робити тільки ті змінні, доступу до яких вимагає більшість функцій програми.

Тепер розглянемо особливості використання локальних змінних. Обидві функції `CreateArray()` та `PrintArray()` працюють зі змінними, що мають ім'я `i`. Проте ці дві змінні повністю незалежні, кожна з них створюється окремо та існує тільки під час виконання своєї функції. Крім `i`, функція `CreateArray()` використовує локальну змінну `n`, оголошену всередині блоку оператора `while`, – її введено спеціально, щоб продемонструвати область дії змінних. У всіх операторах в межах цього блоку використовується значення локальної змінної `n`, а зовні блоку, зокрема в перевірці умови `i<n`, відновлюється дія глобальної змінної `n`. Крім цього, ім'я `n` має формальний параметр у прототипі функції `Contains5()`, але область його дії обмежена даним прототипом. В самому описі функції `Contains5()` відповідний формальний параметр має ім'я `a`, що збігається з іменем глобального масиву. Тому всередині цієї функції діє тільки локальна змінна `a`, яка закриває доступ до глобального масиву з тим самим іменем.

12.2. Специфікатори класів пам'яті

Якщо в описі змінної специфікатор класу пам'яті відсутній, то клас пам'яті такої змінної встановлюється за правилами замовчування. Зазвичай специфікатори застосовують тільки тоді, коли потрібно змінити стандартний клас пам'яті об'єкта. В оголошеннях змінних специфікатор класу пам'яті вказують першим словом:

специфікатор_класу_пам'яті тип ім'я_змінної

Мова С має чотири специфікатори класів пам'яті: `auto`, `register`, `static` та `extern`, всі вони належать до зарезервованих слів. Вплив специфікаторів на область видимості та час існування локальних і глобальних змінних різний.

Специфікатори локальних змінних. В оголошеннях локальних змінних можна використовувати кожен з названих чотирьох специфікаторів (табл. 12.1). За замовчуванням таким змінним присвоюється клас пам'яті `auto` і вони зберігаються в області стека, виділеного для програми в оперативній пам'яті.

Специфікатор `register` початково призначався тільки для змінних з типом `int`. Він вказував, що за наявності вільного системного регістра, змінну слід зберігати в

Таблиця 12.1

Специфікатори класів пам'яті

Специфікатор	Клас пам'яті	Застосування		Час існування	Область дії	Автоматична ініціалізація
		локальні змінні	глобальні змінні			
`auto`	автоматичний	так	ні	локальний	локальна	відсутня
`register`	регістровий	так	ні	локальний	локальна	відсутня
`static`	статичний	так	так	глобальний	локальна / частково-глобальна	ініціалізація нулем
`extern`	зовнішній	так	так	глобальний	частково-глобальна	ініціалізація заборонена

регістрі процесора, а не в оперативній пам'яті, як інші об'єкти. Застосування регістрів дає змогу істотно скоротити час звертання до змінних, тому регістровий клас надають тим змінним, які найбільше впливають на загальну швидкодію програми. Оскільки кількість системних регістрів комп'ютера обмежена, то за відсутності вільних регістрів змінна з класом пам'яті `register` буде опрацьовуватись як звичайна локальна змінна з класом пам'яті `auto`.

Стандарти С-89 і С-99 розширили визначення специфікатора `register`. Тепер він може застосовуватись до змінних різних типів і означає, що доступ до цих змінних повинен бути настільки швидким, наскільки це можливо. Все ж практично найбільший виграш дає застосування регістрового класу до цілочислових змінних, бо компілятор може не підтримувати засобів оптимізації швидкодії для даних інших типів.

Наведемо приклад функції піднесення дійсного числа до цілого степеня, в якій основа, показник степеня та добуток оголошуються як регістрові змінні.

```
/*  функція піднесення дійсного числа до цілого степеня  */
double IntPwr (register double base, register int exp)
{
    register double res;
   for (res=1.0;  exp>0;  exp--)
      res*=base;
   return res;
}
```

Нагадаємо, що до регістрових змінних не можна застосовувати операцію взяття адреси `&`. Якщо певна змінна зберігається в системному регістрі, то її адреса вважається неозначеною.

Застосування специфікатора `static` до локальних змінних змінює час існування цих змінних. Статичні змінні існують протягом усього часу виконання програми, проте областю їх дії залишається той блок, в якому вони оголошені.

Розглянемо ще одну коротку програму.

```
/*********************************************/
/*  Використання статичних змінних у функціях  */
/*********************************************/
#include <stdio.h>
void StaticVarDemo (void)
{
     int a=0;                          /* не ініціалізується автоматично */
     static int b;                     /* автоматично ініціалізується 0 */
   a++;   b++;
   printf (" a=%d   b=%d \n", a, b);
}
int main (void)
{
     int k;
   for (k=0;  k<3;  k++)
      StaticVarDemo();
   return 0;
}
```

Результат виконання програми:

```
a=1   b=1
a=1   b=2
a=1   b=3
```

На відміну від автоматичної змінної `a`, статична змінна `b` ініціалізується тільки один раз – під час першого виклику функції. Після завершення роботи функції змінна `b` не знищується, а її поточне значення зберігається і використовується в наступних викликах функції. Статичні локальні змінні зберігаються в сегменті даних разом з глобальними змінними, але мають обмежену область видимості: змінну `b` можна використовувати тільки всередині функції `StaticVarDemo()`.

Специфікатор `extern` застосовують в оголошеннях локальних змінних, щоб вказати, що ця змінна є посиланням на глобальну змінну з тим самим іменем і типом, описану далі в тексті програми або в іншому програмному файлі. Це робить зовнішню змінну доступною для звертання в межах даного програмного блоку.

```
/*********************************************/
/*  Використання зовнішніх змінних у функціях  */
/*********************************************/
#include <stdio.h>
void ExternVarDemo (void)
{
     extern double x;                  /* посилання на зовнішню змінну */
   printf ("\n x=%.2lf ", x);
   x*=2;
}
```

```
double x=13.246;                    /* опис зовнішньої змінної */
int main (void)
{
   ExternVarDemo();
   ExternVarDemo();
   ExternVarDemo();
   return 0;
}
```

Результат виконання програми:

```
x=13.25
x=26.49
x=39.74
```

Звернемо увагу, що змінна `x`, оголошена в функції `ExternVarDemo()`, не є самостійною змінною, а тільки посиланням на глобальну змінну `x`, яка в тексті програми описана нижче. З цієї причини ініціалізувати локальні змінні зі специфікатором `extern` не можна.

Специфікатори глобальних змінних. В оголошеннях глобальних змінних можна використовувати тільки специфікатори `static` та `extern`. Ці специфікатори найчастіше застосовують, коли створюються багатофайлові програми.

У С-програмах кожна глобальна змінна повинна бути визначена (тобто описана) тільки один раз, а в разі багатофайлових програм – в одному з файлів. Опис змінної пов'язаний із виділенням для неї ділянки пам'яті, обсяг якої визначається типом цієї змінної. Для глобальної змінної пам'ять виділяється в сегменті даних програми і заповнюється нулями, якщо в описі змінну не ініціалізують іншим значенням. В описах змінних зовнішнього рівня або взагалі не використовують специфікаторів, або вказують специфікатор `static`. Застосування класу пам'яті `static` означає, що областю дії даної змінної буде частина програми від точки опису змінної до кінця файла. В інших файлах програми ця змінна недоступна, тому усувається загроза випадкової зміни її значення. Крім того, в інших файлах можна використовувати свої глобальні змінні з тим самим іменем, оскільки всі однойменні змінні зі специфікатором `static` незалежні.

Кожен програмний файл багатофайлової програми може компілюватись автономно. Тому, щоб забезпечити можливість звертання до глобальних змінних, описаних в інших файлах, треба в даному файлі оголосити потрібні глобальні змінні ще раз, використовуючи специфікатор `extern`. Таке оголошення не пов'язане з виділенням пам'яті для змінних, отже змінні зі специфікатором `extern` не можна ініціалізувати – вони є посиланнями на відповідні зовнішні змінні та роблять ці змінні видимими (оголошеними) в межах поточного файла. Таким чином, кожен об'єкт програми повинен бути описаний тільки один раз, але може бути оголошений багаторазово. Найчастіше це стосується прототипів функцій та змінних зі специфікатором `extern`. Для більшості змінних програми описи та оголошення збігаються. Приклад використання специфікаторів класів пам'яті в оголошеннях глобальних змінних наведено в наступному параграфі.

12.3. Багатофайлові програми

У невеликих за розміром програмах функцію `main()` та всі інші функції доцільно записувати в одному текстовому файлі. Якщо ж програма досить велика й складна, то доцільно поділити її на декілька програмних файлів, кожен з яких можна програмувати та компілювати автономно.

У програмах, що поділені на окремі текстові файли, функції розривати не можна. Кожна функція повинна бути повністю записана в одному з програмних файлів.

У багатофайлових програмах питання видимості постають у межах одного файла та на міжфайловому рівні. Встановлено такі правила:

- об'єкти, оголошені на зовнішньому рівні (між функціями), є частково глобальними, а область їх дії простягається від точки оголошення до кінця програмного файла;
- щоб звернутись до глобальної змінної, описаної в іншому файлі, необхідно в поточному файлі оголосити цю змінну зі специфікатором `extern`;
- функції є глобальними об'єктами, тобто вони описуються тільки один раз, а областю їх дії є вся програма;
- прототипи функцій є частково видимими, вони діють у межах одного програмного файла.

Перші два правила розглядались у попередньому параграфі, тому зупинимось тільки на питаннях видимості функцій. Згідно з третім правилом кожна функція повинна бути описана в програмі тільки один раз у якомусь із файлів. Для звертання до функції з інших файлів або перед її описом у тому ж файлі треба попередньо вказати прототип функції. Четверте правило зазначає, що областю дії прототипу функції є один програмний файл, тому в кожному файлі програми треба окремо оголошувати прототипи всіх необхідних функцій. Зокрема, це правило вимагає, щоб відповідні заголовні файли бібліотечних функцій `<*.h>` були підключені до кожного з файлів програми. З користувацькими функціями здебільшого чинять подібно: створюють заголовний файл, в який записують прототипи всіх розроблених функцій загального користування, а потім підключають його через директиву `#include` до кожного з програмних файлів. Унаслідок цього зменшуються розміри текстових файлів і забезпечується доступність розроблених функцій у межах усієї багатофайлової програми.

Наведемо приклад програми, що складається з двох текстових файлів і заголовного файла funchdr.h, в якому записані прототипи користувацьких функцій та декларації шаблонів структурних типів. Перший файл програми містить описи функцій `main()` та `DoFunction()`, а в другому файлі зібрані описи всіх інших функцій. Заголовний файл funchdr.h підключено до обох програмних файлів. Це дає змогу в першому файлі звертатись до функцій `InputData()`, `Func1()` та інших, які описані в другому файлі, а в функціях другого файла опрацьовувати структури, що мають шаблон `OFDATA`.

У першому файлі описано також глобальний масив `arr` і змінну `nd`, а в другому файлі для роботи з ними оголошено посилання на `arr` і `nd`. В обох файлах використовується змінна `k`, яка описана зі специфікатором `static`. Тому насправді це дві незалежні між собою змінні, кожна з них діє тільки в межах свого програмного файла.

```
/*************************************/
/*  Приклад багатофайлової програми  */
/*************************************/

        /* Заголовний файл  funchdr.h */

typedef struct office_data {          /* декларація структурного типу */
  char name [50];
   . . .
} OFDATA;
void DoFunction (void);               /* прототипи функцій */
int InputData (void);
void Func1 (int);
int Func2 (OFDATA*);
   . . .                              /* прототипи решти функцій */

        /* Перший програмний файл */

#include <conio.h>
#include "funchdr.h"
#define NMAX 150
OFDATA arr[NMAX];                     /* глобальний масив */
int nd;                               /* глобальна змінна */
static int k;                         /* частково глобальна змінна */
int main (void)
{
  clrscr();
  while ( k>=0 )
      DoFunction();
  return 0;
}
void DoFunction (void)
{
  switch (k) {
   case 0: nd=InputData();
   case 1: k=3;  Func1(k);  break;
   case 2: k=Func2(arr+NMAX/2);  break;
     . . .
  }
}

        /* Другий програмний файл */

#include <stdio.h>
#include "funchdr.h"
extern OFDATA arr[];                  /* посилання на зовнішній масив */
extern int nd;                        /* посилання на зовнішню змінну */
static int k=15;                      /* змінна другого файла */
```

```
void Func1 (int n)
{
   for (ptr<arr+k)
      puts(; n<nd/2;; n++);
   . . .
   k+=n;
}
int Func2 (OFDATA* ptr)
{
   . . .
   if (ptr<arr+k)
      puts(ptr->name);
   . . .
   return k<nd ? k+1 : -1;
}
int InputData (void)
{
   . . .                              /* заповнення масиву arr */
   return ... ;                       /* повернення кількості елементів */
}
. . .                                 /* описи інших функцій */
```

В інтегрованому середовищі Borland C підтримується робота з багатофайловими програмами. Найбільш простим і гнучким способом компонування єдиного виконавчого коду програми з окремих складових файлів є створення проекту програми. У проекті вказуються імена текстових і/або об'єктних файлів, з яких буде формуватися спільний exe-код програми.

Створення проекту здійснюють через пункт головного меню інтегрованого середовища Project. У вікні діалогу вказують ім'я проекту, а потім наповнюють даний проект іменами складових файлів. Файли проекту можуть бути текстовими (*.c чи *.cpp) або бінарними (*.obj), тобто файлами з об'єктними кодами попередньо відкомпільованих частин програми. Проекти зберігаються як файли з розширеннями *.prj. Створений проект можна редагувати: змінювати його склад чи виправляти тексти програм. За умови відсутності помилок проводиться компіляція текстових файлів проекту, після чого програма-компонувальник формує з отриманого набору об'єктних файлів готовий виконавчий файл (exe-код) програми.

Запитання та завдання для самоконтролю

1. Що визначає клас пам'яті даного? Як він встановлюється?
2. На які групи поділяються об'єкти програми (змінні та функції) за часом існування та областю видимості?
3. Де зберігаються глобальні дані, де – локальні, а де – динамічні?

4. Як встановлюється область дії для глобальних змінних, а як – для локальних? Чи можуть ці змінні в одній програмі мати однакові імена?
5. Які специфікатори класів пам'яті можна використовувати в оголошеннях локальних змінних? Яким є їх вплив на область видимості та час існування цих змінних? Який клас пам'яті отримують локальні змінні за правилами замовчування?
6. Запишіть функцію, яка фіксує скільки разів її викликали і виводить на екран номер виклику. *Підказка*: у функції треба використати статичну локальну змінну.
7. Які специфікатори застосовують для встановлення класів пам'яті глобальних змінних? Яке їх призначення?
8. Якою є область видимості змінних, функцій та їх прототипів у разі багатофайлових програм?
9. Є два програмних файли part1.c та part2.c. У першому файлі зберігаються описи функцій `GetName()` і `SayHello()`, а в другому – описи глобальної змінної `name` та функції `main()`:

```
/*  Файл  part1.c  */
char* GetName (void)
{
  printf("\nВаше ім\'я - ");
  gets(name);
  SayHello();
  return(name);
}
void SayHello (void)
{
    static int nc;
  if (*name)
    printf("\nВітання, %s ", name);
  else
    if (++nc<3)
      GetName();
}
```

```
/*  Файл  part2.c  */
char name[25];
int main (void)
{
  GetName();
  return 0;
}
```

Проаналізуйте кожну функцію. Що вона виконує? Які дані опрацьовує? До яких функцій звертається? Доповніть обидва файли програми необхідними оголошеннями даних і прототипів функцій.

10. Перевірте правильність зроблених у п. 9 доповнень, відкомпілювавши обидва файли part1.c та part2.c.
11. З чого формують проект програми? Як створити з проекту виконавчий файл програми? Чи можна редагувати програмні файли, з яких сформовано проект?
12. З об'єктних файлів, отриманих у результаті виконання п. 10, створіть проект програми hello.prj та виконавчий файл hello.exe. Виконайте програму.

Розділ 13

РОБОТА З ДАНИМИ В ДИНАМІЧНІЙ ПАМ'ЯТІ

У цьому розділі:

- Поняття динамічної пам'яті, переваги динамічного виділення пам'яті для даних програми
- Стандартні бібліотечні функції динамічного виділення/звільнення оперативної пам'яті: `malloc()`, `calloc()`, `realloc()` та `free()`
- Звертання до даних у динамічній пам'яті, приклад застосування статичного масиву вказівників
- Створення динамічного масиву вказівників для збереження багатовимірних масивів у динамічній пам'яті
- Динамічні списки: основні характеристики та способи формування
- Приклад роботи з однозв'язним динамічним списком: формування списку, пошук і видалення заданих елементів, друкування та витирання всього списку
- Структура елемента двозв'язного списку; формування впорядкованого списку, доповнення списку новими елементами, видалення окремих елементів та інші прийоми роботи з двозв'язним списком; приклад програми
- Двійкові дерева: конфігурація, впорядкованість, збалансованість; обхід двійкового дерева; рекурсивні методи програмування функцій, призначених для роботи з двійковими деревами
- Приклад програми формування та опрацювання двійкового дерева пошуку
- Нерекурсивний варіант функції обходу дерева

У попередньому розділі зазначалось, що дані, які в програмі мають глобальний час існування, зберігаються в окремій області оперативної пам'яті, виділеній для статичних даних програми; дані з локальним часом існування зберігаються в області пам'яті, яку називають стеком; а т. зв. *динамічні дані* – в області пам'яті, що підлягає динамічному розподілу. Область динамічних даних – це вільна частина оперативної пам'яті, не зайнята на даний момент програмою, операційною системою чи іншими

активними програмами. Цю область оперативної пам'яті часто називають *heap*-пам'яттю (чи просто *Heap*, що в перекладі означає купа або нагромадження), ми ж будемо вживати термін *динамічна пам'ять*.

Збереження даних у динамічній пам'яті має декілька важливих переваг:

- виділенням і звільненням динамічної пам'яті керує програма, тому пам'ять для даних можна виділяти тільки на час опрацювання цих даних, а потім звільняти її для інших потреб;
- обсяг пам'яті, виділеної для динамічних даних, теж задається програмно, тому його можна встановлювати строго відповідним до реального розміру даних – це насамперед важливо для масивів і символьних рядків, поточні розміри яких найчастіше стають відомими тільки під час виконання програми;
- за необхідності в процесі роботи можна змінити (зменшити або збільшити) обсяг ділянки динамічної пам'яті, яку займає дане;
- здебільшого вільна динамічна пам'ять достатньо велика – її обсяг перевищує обсяги сегментів статичних даних і стека, тому в динамічній пам'яті можна розташовувати великі масиви та інші об'єкти, які не вміщаються у сегменті даних чи в стеку;
- динамічне виділення пам'яті дає змогу створювати такі ефективні та гнучкі інформаційні конструкції, як динамічні структури даних.

13.1. Стандартні функції динамічного виділення пам'яті

Стандарт мови С підтримує чотири функції, призначені для звертання до динамічної пам'яті: `malloc()`, `calloc()`, `realloc()` та `free()`. Прототипи перелічених функцій оголошені в стандартному заголовному файлі `<stdlib.h>` (а також у заголовному файлі `<alloc.h>`, який не стандартизований).

Для роботи з динамічною пам'яттю більшість компіляторів підтримує, крім названих вище, інші спеціальні функції, що базуються на апаратно-програмних особливостях конкретної операційної платформи. Так, у системі програмування Borland C функції звертання до динамічної пам'яті (їх прототипи оголошені в `<alloc.h>`) враховують особливості різних моделей пам'яті, застосовують відповідно короткі або довгі вказівники, дають змогу визначити обсяг незайнятої динамічної області тощо. Ці функції розглянемо в розділі 14. Наразі просто рекомендуємо компілювати програми, в яких використовується динамічна пам'ять, встановивши модель розподілу пам'яті Large. Вибір моделі пам'яті здійснюють через пункт меню інтегрованого середовища Options/Compiler/Code generation/Model.

Функція `malloc()`. Основною функцією виділення динамічної пам'яті є функція

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

Функція `malloc()` має один параметр `msize`, що задає обсяг (у байтах) неперервної ділянки, яка повинна бути виділена в динамічній пам'яті. Тип `size_t` використовується в оголошеннях багатьох функцій мови С. Його введено (задекларовано через `typedef` у відповідних заголовних файлах), щоб забезпечити універсальність і мобільність ого-

лошень параметрів розмірів (обсягів пам'яті). Як правило, `size_t` відповідає одному з цілих беззнакових типів: `unsigned int` або `unsigned long` (зокрема, в Borland C тип `size_t` рівнозначний типу `unsigned int`). За умови успішного виконання функція `malloc()` повертає адресу першого байта ділянки заданого обсягу, виділеної у динамічній пам'яті. Якщо ж вільної пам'яті недостатньо, то функція повертає `NULL` (значення макроконстанти, якою позначають порожній вказівник).

Подамо приклад використання `malloc()` для виділення динамічної пам'яті.

```
/*  Виділення у динамічній пам'яті ділянки заданого обсягу */
#include <stdlib.h>
. . .
   int  *pm, dsize = 600;
pm = malloc(dsize);
if (pm == NULL) {                          /* можна записати  if (!pm) */
   puts("Відсутня вільна пам\'ять!");
   . . .                    /* інші дії, пов'язані з нестачею пам'яті */
}
```

Для наведеного прикладу обсяг ділянки, виділеної в динамічній пам'яті, становитиме 600 байтів, тобто в цю ділянку можна занести масив із 300 даних, що мають тип `int`. Загалом у виділеній ділянці можна розмістити `dsize/sizeof(*pm)` елементів.

Слід пам'ятати, що обсяг вільної динамічної пам'яті змінний, він залежить від складу системних і користувацьких програм, які на даний момент використовують оперативну пам'ять комп'ютера.

У разі виділення великих за обсягом динамічних ділянок може виникнути ситуація нестачі вільної heap-пам'яті. Тому, здійснюючи динамічне виділення пам'яті для даних програми, треба обов'язково перевірити результат виконання відповідної функції і передбачити реакцію програми на випадок, коли не вдасться виділити необхідну за обсягом пам'ять.

Функція `malloc()` повертає вказівник з базовим типом `void`. Мова C дозволяє присвоювати адресу, оголошену як `void`-вказівник, вказівникам на дані всіх типів. Присвоєння `pm=malloc(dsize)` коректне та пов'язане з автоматичним перетворенням типу `void*` до типу `int*`.

У програмах мовою C++ необхідно виконувати явне перетворення типу вказівника, тобто попереднє присвоєння слід записувати так:

```
pm = (int*)malloc(dsize);
```

Для сумісності з C++ надалі будемо явно перетворювати тип значення всіх функцій виділення динамічної пам'яті, хоча для C-програм таке перетворення не обов'язкове.

Функції `calloc()` та `realloc()`. Іншою функцією, призначеною для виділення динамічної пам'яті, є `calloc()`:

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

Ця функція виділяє в області динамічних даних неперервну ділянку, достатню для розташування масиву з num елементів, кожен з яких має розмір size. Результат виділення збігається з результатом виклику функції malloc(num*dsize). Особливість виконання функції calloc() у тому, що всі біти виділеної пам'яті заповнюються нулями (тобто ділянка онулюється). В усьому іншому ця функція рівнозначна функції malloc().

Наведена нижче функція GetArrayMem() виділяє динамічну пам'ять для масиву з k дійсних чисел і повертає вказівник на його перший елемент.

```
/*  Виділення динамічної пам'яті для масиву дійсних чисел  */

#include <stdlib.h>

double*  GetArrayMem (int k)
{
     double *arr;
  arr = (double*)calloc(k, sizeof(double));
  if (!arr) {
     . . .                          /* дії у разі нестачі пам'яті */
  }
  return arr;
}
```

Функція realloc() призначена для зміни обсягу ділянки динамічної пам'яті, попередньо виділеної функціями malloc() або calloc(). Прототип її такий:

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

тут ptr – вказівник на ділянку динамічної пам'яті, обсяг якої треба змінити (збільшити або зменшити); newsize – новий обсяг ділянки в байтах.

У разі успішного завершення realloc() повертає вказівник на ділянку нового обсягу, а в разі невдачі – NULL. Стара ділянка динамічної пам'яті в останньому випадку не змінюється.

Дія функції realloc() за стандартами С-89 і С-99 дещо відмінна. За стандартом С-89 зміна обсягу ділянки необов'язково вимагає перенесення даних (зокрема в разі зменшення ділянки), але й не виключає його. Якщо ж виділяється нова ділянка динамічної пам'яті, то дані зі старої області автоматично копіюються в нову. За стандартом С-99 функція realloc() звільняє ділянку, на яку вказує ptr, і переносить у нововиділену ділянку вміст старої ділянки (або тільки її початкові newsize байт).

Якщо є цілковита впевненість, що зміна обсягу ділянки відбудеться успішно, то її можна запрограмувати, наприклад, так:

```
pm = (int*)realloc(pm, dsize/2);
```

У результаті виконання realloc() обсяг ділянки пам'яті, виділеної раніше функцією malloc(), буде зменшено вдвічі.

Якщо ж імовірно, що вільної пам'яті може бути недостатньо для розширення чи перенесення ділянки, то попередня форма виклику realloc() загрожує тим, що вказівник pm може набути значення NULL, а це спричинить втрату зв'язку зі старою

ділянкою пам'яті. Найкраще для зміни обсягу динамічної ділянки використовувати додатковий робочий вказівник, як це показано в наступному прикладі.

```
/*  Зміна обсягу ділянки, розташованої в динамічній пам'яті  */
  int* newmem;
newmem = (int *)realloc(pm, dsize*2);
if (newmem)              /* тобто newmem!=NULL - нову ділянку створено */
   pm = newmem;
else {
   puts("Недостатньо пам\'яті!");
   . . .                 /* інші дії, пов'язані з нестачею пам'яті */
}
```

Якщо першим параметром функції `realloc()` є `NULL`, то відбувається звичайне виділення ділянки динамічної пам'яті заданого обсягу, як це виконує `malloc()`. Коли ж нулю дорівнює параметр нового обсягу ділянки, то вказана ділянка звільняється.

Функція `free()`. Для звільнення ділянки динамічної пам'яті, виділеної раніше однією із функцій: `malloc()`, `calloc()` чи `realloc()`, застосовують функцію

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

Ця функція повертає в область пам'яті, що підлягає динамічному розподілу, ділянку, на початок якої вказує `ptr`. Надалі цю ділянку зможуть використовувати функції виділення динамічної пам'яті.

```
/*  Звільнення ділянки в динамічній пам'яті  */
free (pm);
```

Необхідно стежити, щоб параметр функції `free()` дійсно був адресою ділянки, попередньо виділеної в динамічній пам'яті. У разі помилкової адреси результат роботи функції не визначений, він може серйозно пошкодити всю систему керування динамічною пам'яттю.

13.2. Використання масивів вказівників на динамічні дані

Популярним способом організації зв'язків з даними, розташованими в динамічній пам'яті, є створення масиву вказівників, кожен елемент якого зберігає адресу одного об'єкта, записаного в динамічну пам'ять.

13.2.1. Приклад застосування статичного масиву вказівників

Розглянемо задачу. Нехай з клавіатури вводиться певний текст – припустимо, що він складається з окремих речень (символьних рядків). Треба доповнити кожен введений рядок числовим значенням, яке дорівнює кількості слів у даному реченні.

Оскільки довжини рядків тексту будуть різними, а їх кількість буде змінюватись залежно від конкретної реалізації, то доцільно зберігати введені рядки в динамічній пам'яті. Для кожного рядка треба виділити пам'ять, що відповідає розміру рядка з урахуванням нуль-символа і додаткових 5 байтів для запису кількості слів. Введений рядок спочатку заноситься у буфер введення `buf`, а звідти переписується у виділену ділянку динамічної пам'яті.

Адреси рядків, розташованих у динамічній пам'яті, зберігатимемо в масиві вказівників `strpar`, який оголосимо як глобальний. Для звертання до елементів масиву `strpar` будемо використовувати як індексні вирази, так і окремий вказівник `parp`. Цей вказівник повинен вказувати на елементи масиву вказівників, що вказують на перші символи рядків. Отже, його треба оголосити так:

```
char** parp;
```

Значенням виразу `*parp` буде адреса динамічної пам'яті, за якою записано відповідний символьний рядок. Оскільки масив `strpar` глобальний, то всі його елементи, незаповнені адресами рядків, будуть проініціалізовані нульовими значеннями. Тому, просуваючи по масиву вказівник `parp`, можна не рахувати кількість елементів, а тільки перевіряти, чи не досягнуто елемента зі значенням `NULL`.

Для визначення кількості слів у реченні та запису обчисленого значення у кінець символьного рядка використаємо дві функції: `WordsNumb()` та `AddWrdNumb()`. Це дасть змогу розвантажити `main()` і наочніше виділити операції взаємодії з динамічною пам'яттю.

Після доповнення рядків тексту значенням кількості слів і виведення на екран усіх доповнених рядків звільняємо динамічну пам'ять, зайняту введеним текстом.

```
/*************************************************************/
/*  Опрацювання символьних рядків у динамічній пам'яті (ДП)  */
/*************************************************************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define LEN 150                          /* максимальна довжина рядка */
#define NSTR 100                         /* максимальна кількість рядків */
int  WordsNumb (char* st);               /* прототипи функцій */
char*  AddWrdNumb (char* st, int n);
char* strpar[NSTR];                       /* глобальний масив вказівників */
int  main()
{
    char buf[LEN], *pmem, **parp;
    int nstr, nw;
/*  Занесення символьних рядків у динамічну пам'ять  */
  puts("\n\tТекст:");
  nstr = 0;
```

```
  while ( *gets(buf) && nstr < NSTR-1 ) {/* поки не введено порожній рядок */
     pmem =(char *)malloc(strlen(buf)+6);      /* виділення місця в ДП */
     if ( pmem == NULL ) {                        /* недостатньо місця */
        puts ("Відсутня вільна пам\'ять");
        break;                                        /* кінець введення */
     }
     strcpy(pmem, buf);                        /* перенесення рядка в ДП */
     strpar[nstr++]= pmem;                  /* запис адреси рядка в масив */
  }
/*  Доповнення рядків значенням кількості слів  */
  puts("\tДоповнені речення:");
  for ( parp = strpar;  *parp != NULL;  parp++) {      /* цикл по рядках */
     nw = WordsNumb(*parp );        /* визначення кількості слів у рядку */
     puts( AddWrdNumb(*parp, nw));     /* доповнення і виведення рядка */
  }
/*  Звільнення виділеної динамічної пам'яті  */
  parp = strpar;
  while (*parp != NULL);        /* цикл по елементах масиву вказівників */
     free(*parp++);                          /* витирання рядків в ДП */
  return 0;
}
/*  Функція, що визначає кількість слів у заданому рядку  */
int  WordsNumb (char* str)
{
    int kw=1;                              /* у рядку є хоча б одне слово */
    char * ps = str;
  while (*ps != '\0') {                        /* цикл по символах рядка */
     if(*ps == ' ' && *(ps+1)!= ' ')                  /* початок слова */
        kw++;
     ps++;
  }
  return kw;
}
/*  Функція, що приєднує до речення значення кількості слів  */
char*  AddWrdNumb (char* str, int numb)
{
    char * ps = str;
  while (*ps != '\0')  ps++;                    /* перехід у кінець рядка */
  *ps++ = ' ';  *ps++ = '-';  *ps++ = ' ';            /* долучення " - " */
  if (numb < 10)                             /* кількість слів одноцифрова */
     *ps++ = numb+'0';              /* долучення значення кількості слів */
  else {                                              /* запис кількості  */
     *ps++ = numb/10+'0';  *ps++ = numb%10+'0';    /* двоцифрових слів */
  }
  *ps = '\0';                                /* кінець доповненого рядка */
  return str;
}
```

Приклад виконання:

```
    Текст:
Настав вечір.
Сонце, стомлене довгим трудовим днем, котилося до обрію.
Нарешті повіяло прохолодою.
З-над ріки долинув дівочий спів.
 < порожній рядок, що є ознакою кінця введення >
    Доповнені речення:
Настав вечір. - 2
Сонце, стомлене довгим трудовим днем, котилося до обрію. - 8
Нарешті повіяло прохолодою. - 3
З-над ріки долинув дівочий спів. - 5
```

13.2.2. Створення динамічного масиву вказівників

Очевидним недоліком попередньої програми є фіксована розмірність масиву вказівників. З одного боку, це не дає змоги опрацьовувати більше, ніж `NSTR-1` рядків. З іншого боку, якщо встановити значення `NSTR` достатньо великим, то це призведе до неефективного використання оперативної пам'яті, оскільки в більшості реалізацій програми кількість введених рядків буде значно меншою за `NSTR`.

Обмеження, пов'язані з фіксованою розмірністю масиву вказівників, можна усунути, якщо масив вказівників також формувати динамічно, встановлюючи його розмір таким, як цього вимагає кожна конкретна реалізація. Якщо в процесі виконання програми виявиться, що розмір масиву вказівників потрібно змінити, то це можна легко запрограмувати, використовуючи стандартну функцію `realloc()`.

Розглянемо ще одну задачу. Необхідно сформувати прямокутну матрицю вказаної розмірності, заповнивши її випадковими дійсними числами зі заданого діапазону значень. Рядки заповненої матриці треба переставити так, щоб суми їхніх елементів утворювали спадну послідовність.

Оскільки реальна розмірність матриці стає відомою тільки в процесі виконання програми, то доцільно розташувати матрицю в динамічній пам'яті. Тоді для зберігання матриці можна виділити ділянку, обсяг якої буде строго відповідати кількості даних у конкретній реалізації програми. До складу кожного рядка матриці долучимо додатковий елемент, у який занесемо значення суми цього рядка. Завдяки цьому уникнемо кількаразового обчислення суми елементів у процесі сортування рядків.

Щоб скоротити час сортування рядків матриці та спростити звертання до її елементів, створимо масив вказівників на початки рядків матриці. Тепер замість переставляння самих рядків, що вимагає попарного обміну значеннями всіх елементів двох рядків, які міняються місцями, у процесі сортування можна використати прийом переставляння вказівників на відповідні рядки (графічна ілюстрація цього прийому наводилась на рис. 9.3 і 9.4). Після сортування рядки матриці залишаються на своїх місцях у динамічній пам'яті, а вказівники на ці рядки переставляються у масиві так, щоб їх послідовність вказувала на рядки в порядку спадання значень сум їхніх елементів.

Наведемо повний текст програми з основними коментарями, а потім детальніше проаналізуємо програмні особливості окремих функцій.

```
/*****************************************************************/
/*  Формування в динамічній пам'яті (ДП) матриці, заповненої     */
/*  дійсними випадковими числами та впорядкованої за спаданням   */
/*  значень сум елементів рядків                                 */
/*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
double ** CreateArray (void);                          /* прототипи функцій */
void PrintArray (double * matr[], int print_sum);
void SortRows (double ** matr);
void DeleteArray (double ** matr, int rdel);
int nr, nc;                        /* кількість рядків і стовпців матриці */
int  main (void)
{
    double * *parrp;                        /* вказівник на елементи масиву
                                               вказівників, розміщеного в ДП */
  parrp = CreateArray();                              /* формування матриці */
  if ( !parrp ) {                                /* недостатньо місця в ДП */
    puts ("Недостатньо пам\'яті");
    return 0;
  }
  puts ("\n\t\t Сформована матриця:");
  PrintArray ( parrp, 0);                    /* виведення сформованої матриці */
  SortRows ( parrp );                          /* сортування рядків матриці */
  puts ("\n\t\t Відсортована матриця:");
  PrintArray ( parrp, 1);    /* виведення результату сортування з сумами */
  DeleteArray ( parrp, nr);                                /* звільнення  ДП */
  return 0;
}
/*  Функція формування прямокутної матриці розмірністю nr×nc,
    заповненої випадковими дійсними числами зі заданого діапазону  */
double * *  CreateArray (void)
{
    double **pmat;          /* вказівник на елементи масиву вказівників */
    int k, j;
    int min, max, intp;
    double fracp;
  printf ("  Розмірність матриці -  ");
  scanf ("%d%d", &nr, &nc);
  pmat = (double**)calloc (nr, sizeof(double *));      /* виділення ДП для
                          масиву вказівників на початки рядків матриці */
  if ( !pmat )                                   /* недостатньо місця в ДП */
       return NULL;
```

```
  printf("  Діапазон значень елементів -  ");
  scanf("%d%d", &min, &max);
  randomize();
  for (k=0;  k<nr;  k++) {                          /* формування матриці */
    pmat[k]=(double *)calloc(nc+1, sizeof(double));     /* виділення ДП
                                                       для k-го рядка матриці */
    if ( !*(pmat+k) ) {                            /* недостатньо місця в ДП */
       DeleteArray(pmat, k);
       return NULL;
    }
    for (j=0;  j<nc;  j++) {                  /* заповнення рядків числами */
       intp=min+random(max-min);                    /* ціла частина числа */
       fracp=(double)rand()/RAND_MAX;             /* дробова частина числа */
       pmat[k][j]=intp+fracp;                    /* запис сформованого числа */
    }
  }
  return pmat;
}
/*  Функція виведення матриці (та сум її рядків, якщо print_sum==1)  */
void PrintArray (double * prow[], int print_sum)
{
    double **pr, *pel;
    int k,j;
  for (k=0, pr=prow;  k<nr;  k++, pr++) {
    for (j=0, pel=*pr;  j<nc;  j++, pel++)
      printf("%9.2lf", *pel);                  /* виведення елементів рядка */
    if (print_sum)
      printf("%14.3lf", *(*pr+nc));                 /* виведення суми рядка */
    printf("\n");
  }
}
/*  Функція сортування рядків матриці за спаданням значень їхніх сум  */
void SortRows (double** mrowp)
{
    double **pr, **pr1, **pr2;
    double *pel, *ps;
  for (pr=mrowp;  pr<mrowp+nr;  pr++)          /* цикли обчислення сум */
    for (pel=*pr, ps=*pr+nc;  pel<ps;  pel++)                 /* рядків */
      *ps+=*pel;          /* нагромадження суми в останньому елементі */
  for (pr1=mrowp;  pr1<mrowp+nr-1;  pr1++)      /* сортування рядків */
    for (pr2=pr1+1; pr2<mrowp+nr;  pr2++)
      if ((*pr1)[nc]<(*pr2)[nc]) {                        /* порівняння сум */
          ps=*pr1;                          /* обмін значеннями вказівників, */
          *pr1=*pr2;                        /* що містять адреси рядків, які */
          *pr2=ps;                                /* мають бути переставлені */
      }
}
```

```
/*  Функція звільнення виділеної динамічної пам'яті  */
void  DeleteArray (double ** ppar, int rdel)
{
    double **pr;
  for (pr = ppar;  pr < ppar+rdel;  pr++)
     free( *pr );                                    /* витирання рядка */
  free( ppar );                              /* витирання масиву вказівників */
}
```

Приклад виконання:

```
Розмірність матриці - 5 6
Діапазон значень елементів - 0 500

          Сформована матриця:
 356.61   488.70   148.54   247.03     7.33   320.23
 101.20   244.98   168.90   376.51   224.15   490.18
 215.62   430.83   124.81   413.26   250.63   338.18
 144.13   227.99   447.06   142.66   106.83   191.23
 201.43   492.97   376.89   185.98    99.08   378.05

          Відсортована матриця:
 215.62   430.83   124.81   413.26   250.63   338.18     1773.334
 201.43   492.97   376.89   185.98    99.08   378.05     1734.403
 101.20   244.98   168.90   376.51   224.15   490.18     1605.917
 356.61   488.70   148.54   247.03     7.33   320.23     1568.441
 144.13   227.99   447.06   142.66   106.83   191.23     1259.899
```

Вищенаведена програма демонструє різні способи звертання до елементів матриці, розташованої в динамічній пам'яті: через індекси, через вказівник на елементи рядка, за допомогою окремого вказівника на початки рядків тощо.

Зупинимось на ключових моментах функцій програми. Функція `CreateArray()` формує в динамічній пам'яті матрицю, кожен елемент якої є випадковим дійсним числом. Число формується з двох частин: цілої `intp`, діапазон значень якої встановлює користувач, та дробової `fracp`, яка обчислюється на основі стандартної бібліотечної функції `rand()`, що повертає ціле випадкове число з діапазону `0..RAND_MAX`:

```
fracp = (double)rand()/RAND_MAX;
```

Для ефективної роботи з рядками матриці першим у динамічній пам'яті формується масив вказівників, кожен елемент якого зберігатиме адресу відповідного рядка матриці:

```
pmat = (double**)calloc(nr, sizeof(double*));
```

Змінна `pmat` виконує роль вказівника на початок масиву вказівників на перші елементи рядків матриці. Тому цю змінну оголошено з типом `double**`. Масив вказівників заповнюється адресами рядків матриці. До кожного рядка долучаємо один додатковий елемент, в якому зберігатиметься сума цього рядка:

```
*(pmat+k) = (double*)calloc(nc+1, sizeof(double));
```

В обох випадках динамічна пам'ять виділяється за допомогою стандартної функції `calloc()`, яка на відміну від функції `malloc()`, відразу онулює виділену область динамічної пам'яті.

Функцію виведення матриці `PrintArray()` організовано так, що можна виводити тільки елементи матриці (для цього другий параметр функції повинен дорівнювати `0`) або додатково в кінці кожного рядка виводити значення суми елементів цього рядка (за умови, що другий параметр дорівнює `1`).

Функція `SortRows()` переставляє рядки матриці за спаданням значень їхніх сум. Фактично відбувається обмін значеннями між елементами масиву вказівників. Після закінчення циклу сортування послідовність вказівників буде вказувати на рядки в тому порядку, що відповідає умові спадання значень сум. Для сортування використано додаткові вказівники `pr1` та `pr2`, які є вказівниками на елементи масиву вказівників, що, зберігають адреси тих рядків, суми яких порівнюються:

```
if ((*pr1)[nc]<(*pr2)[nc]) {      /* порівняння значень сум рядків */
   pel = *pr1;                    /* обмін значеннями вказівників, */
   *pr1 = *pr2;                   /* що зберігають адреси рядків,  */
   *pr2 = pel;                    /* які мають бути переставлені   */
}
```

Остання функція програми `DeleteArray()` призначена для звільнення динамічної пам'яті, яку займали дані програми. Спочатку витираються заповнені рядки матриці, а потім звільняється пам'ять, виділена для масиву вказівників на початки рядків.

13.3. Динамічні списки

Хоча застосування статичних або динамічних масивів вказівників є зручною формою збереження адрес даних, які розташовані в динамічній пам'яті, все ж такі масиви доцільно використовувати у випадках, коли кількість і конфігурація даних заздалегідь відома та не змінюється у процесі виконання програми.

Проте в багатьох задачах кількість даних наперед не встановлена, а розв'язування задачі передбачає введення нових елементів у вже сформований набір даних, видалення окремих елементів чи інші зміни складу й конфігурації набору даних. У цих випадках вищої гнучкості та ефективності в програмуванні й реалізації задач можна досягти, застосовуючи т. зв. *динамічні інформаційні структури*, найбільш поширеними формами яких є *лінійні списки* та *двійкові дерева* (розгалужені списки).

Динамічні інформаційні структури формуються на основі взаємозв'язку елементів. Так, кожен елемент лінійного списку пов'язується через відповідні вказівники з одним або декількома сусідніми елементами (рис. 13.1). Програмно елемент динамічного списку можна реалізувати через структурний тип, який містить щонайменше два поля: інформаційне, в якому зберігаються дані цього елемента, та адресне, через яке він пов'язується з іншим елементом. Здебільшого всі елементи списку мають однаковий структурний тип.

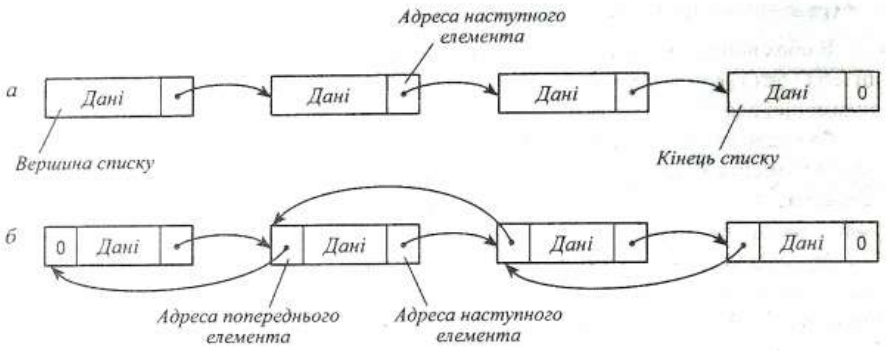


Рис. 13.1. Динамічні списки: *а* – однозв'язний список; *б* – двозв'язний список

Лінійний список, кожен елемент якого містить посилання на один сусідній (як на рис. 13.1, *а*), називається *однозв'язним* або *однонапрямленим*. Поширені також *двозв'язні* або *двонапрямлені* лінійні списки, в яких кожен елемент зберігає адреси двох сусідніх (рис. 13.1, *б*).

Оскільки елементи списку формують ланцюг даних, по якому можна рухатись в одному (однозв'язний список) чи обох (двозв'язний список) напрямах, то для звертання до списку достатньо зберігати адресу тільки одного початкового елемента, який називають *вершиною* або *головою* списку. Кінцевий елемент, який не має наступників, називають *хвостом* списку, а в його адресне поле заносять константу `NULL`.

Список може бути *замкненим* (*кільцевим*), коли кінцевий елемент містить посилання на вершину списку – фактично такі списки не мають початку й кінця. Для зв'язку з кільцевим списком можна зберігати адресу довільного елемента цього списку.

Списки розрізняють також і за способом приєднання нових елементів. Найбільш поширені такі способи доповнення списку:

- приєднання елементів до вершини списку;
- приєднання елементів до кінця списку;
- вставлення елементів у визначені позиції списку.

Перший спосіб передбачає, що кожен новий елемент приєднується перед початковим і стає в списку першим (вершиною списку). Таку організацію даних називають списком, організованим як *стек*. При проходженні по такому списку від вершини до кінцевого елемента дані зчитуються у порядку, зворотному до порядку їх запису.

За другим способом формується список, який називають *чергою*, оскільки кожен новий елемент стає останнім (кінцевим) елементом списку.

Третій спосіб дає змогу формувати списки, впорядковані за певними правилами, наприклад, впорядковані за значеннями ключів інформаційних полів (*ключем* називають певну ознаку, за якою здійснюється впорядковування і/або пошук елементів). Новий елемент може вводитись у довільну позицію списку, зокрема всередину між двома вже записаними елементами.

13.3.1. Однозв'язні списки

Основні прийоми роботи з однозв'язними динамічними списками розглянемо на прикладі такої задачі. Припустимо, що користувач послідовно вводить з клавіатури дані, які складаються з індекса та текстового повідомлення. Із введених даних у динамічній пам'яті необхідно сформувати список, організований як черга, і надрукувати (вивести на екран) його інформаційні поля у формі таблиці. Потім зі списку треба видалити всі елементи, які мають непарні індекси, та надрукувати список, що залишився.

За умовою задачі дані, що будуть заноситись в інформаційне поле кожного елемента списку, мають дві складові частини: індекс і текстове повідомлення. Тому для їх оголошення введемо окремий структурний тип:

```
typedef struct inform {
     int index;
     char message[розмір_повідомлення];
} INFORM;
```

Тоді елемент списку буде структурою, шаблон якої можна оголосити так:

```
typedef struct list_elem {
     INFORM inform;
     struct list_elem *next;
} LEL;
```

тут `inform` – поле даних елемента, а `next` – вказівник на наступний елемент списку.

Адресу початку списку будемо зберігати в змінній-вказівнику `list`, яку для спрощення функцій програми оголосимо як глобальну змінну:

```
LEL* list;
```

Нагадаємо, що всі глобальні змінні автоматично ініціалізуються нульовим значенням, тобто вказівник `list` отримає значення `NULL`, яким відзначається порожній список.

Формування списку. Робота програми розпочинається з введення даних і формування списку. Оскільки список має бути сформований як черга, то кожен нововведений елемент заноситиметься у кінець списку (ставатиме останнім). Ознакою завершення процесу введення даних будемо вважати появу нульового індекса.

Запишемо окрему функцію `AddElem()`, призначену для введення даних (індекса й текстового повідомлення), формування з цих даних нового елемента списку і приєднання його до кінця списку. Єдиним параметром функції є вказівник `last`, через який передається адреса останнього елемента попередньо сформованої черги. До цього елемента буде приєднаний новий, створений у функції. Якщо список ще не містить даних (значення `list` дорівнює `NULL`), то нововведений елемент стає не тільки останнім, але й першим елементом черги, а його адреса заноситься у глобальний вказівник на початок списку `list`. Для нумерації введених даних у функції використано статичну змінну `num`. `AddElem()` повертає адресу приєднаного елемента або `NULL`, якщо введення даних завершено (зчитано нульовий індекс).

```
LEL * AddElem (LEL * last)
{
    LEL *pel;                                  /* вказівник на новий елемент */
    static int num = 1;                        /* номер елемента, що вводиться */
  pel = (LEL*)malloc(sizeof(LEL));             /* виділення ДП для елемента */
  printf ("\n\t%d елемент:  індекс - ", num);
  scanf ("%d", &pel->inform.index);
  if (pel->inform.index == 0) {                          /* кінець введення */
    free (pel);
    return NULL;
  }
  fflush (stdin);                              /* очищення буфера введення */
  printf ("Повідомлення:  ");
  gets (pel->inform.message);
  pel->next = NULL;                          /* елемент стає останнім у списку */
  if (list == NULL)                              /* якщо список порожній, то */
     list = pel;                           /* введений елемент стає першим */
  else
     last->next = pel;                  /* приєднання до останнього в списку */
  num++;
  return pel;
}
```

Зупинимось детальніше на формуванні елемента динамічного списку. Оператор

```
pel = (LEL *)malloc(sizeof(LEL));
```

звертається до бібліотечної функції `malloc()`, щоб виділити ділянку динамічної пам'яті, відповідну за обсягом до розміру елемента списку. Адреса ділянки нового елемента присвоюється вказівнику `pel`.

Щоб скоротити тексти програм і зосередити увагу на особливостях їх алгоритмів, опускатимемо перевірку наявності вільної динамічної пам'яті. Проте в усіх практичних програмах, що використовують для збереження даних динамічну пам'ять, таку перевірку треба здійснювати обов'язково.

Коли динамічну пам'ять для елемента виділено, в поля внутрішньої структури `inform` заносяться введені з клавіатури дані:

```
scanf ("%d", &pel->inform.index);   gets (pel->inform.message);
```

У поле адреси наступного елемента – `next` записується значення `NULL`, оскільки новий елемент стає останнім у черзі.

Виведення на екран і витирання всього списку. Зв'язок зі створеним списком здійснюється через вказівник на початок списку `list`, який оголошено як глобальну змінну. Щоб роздрукувати інформаційні поля всіх наступних елементів списку, треба послідовно переходити з одного елемента на інший, адреса якого задається полем `next`. Розглянемо функцію друкування списку `PrintList()`.

```
void PrintList (void)
{
    LEL * pel = list;
  while (pel!= NULL) {
    printf ("\n%-8d%-70s", pel->inform.index, pel->inform.message);
    pel = pel-> next;
  }
}
```

Для звертання до елементів динамічного списку функція `PrintList()` використовує вказівник `pel`. Оператор

```
pel = pel-> next;
```

пересуває цей вказівник по елементах списку, тобто встановлює `pel` на перший байт наступного елемента списку, адреса якого зберігається в полі `next` поточного динамічного елемента, на який вказує `pel`. Цикл завершується, коли виведено дані останнього елемента, оскільки в його поле `next` записано константу `NULL`.

Функція витирання списку `FreeList()` подібна до описаної вище.

```
void FreeList (void)
{
    LEL * pel = list;
  while (pel!= NULL) {
    list = list-> next;      /* першим у списку стає наступний елемент */
    free (pel);              /* витирання поточного елемента */
    pel = list;
  }
}
```

Для процесу витирання важливий порядок операторів: спочатку вказівник `list` переноситься на наступний елемент списку та робить його вершиною списку, а потім витирається попередній елемент, на який вказує `pel`. Якщо ці оператори переставити місцями:

```
free (pel);
list=list->next;
```

то в другому операторі відбуватиметься хибне звертання до поля `next` того елемента, який вже видалено.

Розроблені функції формування, виведення та витирання списку реалізують першу частину поставленої задачі. Доповнимо їх функціями, що виконують пошук елементів за заданими ознаками та видаляють знайдені елементи зі списку.

Пошук і видалення окремих елементів. Функція `FindOddIndex()` призначена для пошуку в списку першого елемента з непарним індексом, вона повертає вказівник на знайдений елемент або `NULL`. Пошук проводиться, починаючи з елемента списку, на який вказує параметр функції `start`. Це дає змогу перевіряти тільки ту частину списку, яка ще не розглядалась.

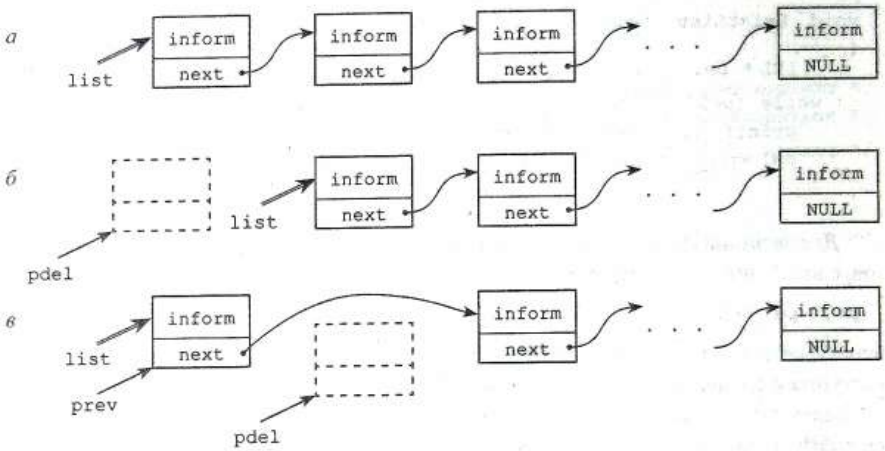


Рис. 13.2. Видалення елемента однозв'язного списку:
а – сформований список;
б – видалення першого елемента;
в – видалення серединного елемента

```
LEL * FindOddIndex (LEL * start)
{
    LEL * pel = start;          /* пошук починається з елемента *start */
  while (pel!= NULL)
    if ( pel->inform.index % 2 != 0 )       /* індекс елемента непарний */
       return pel;
    else
       pel = pel-> next;
  return NULL;                   /* не знайдено відповідного елемента */
}
```

Функція `DeleteElem()` видаляє зі списку елемент, на який вказує `pdel`. Якщо видаляється перший елемент списку, то вказівник початку списку `list` переноситься на наступний елемент (рис. 13.2, *б*). Видалення всіх інших елементів вимагає збереження зв'язності списку. Для цього в елемент, який передував `pdel`, треба занести адресу елемента, наступного за видаленим (рис. 13.2, *в*), або `NULL` у разі видалення останнього елемента списку. Проте в однозв'язних списках зберігається тільки адреса наступного елемента, тому функція має другий параметр – вказівник `prev`, що визначає частину списку, в якій треба шукати попередника `pdel` (тобто елемент, в полі `next` якого записано адресу `pdel`).

```
LEL * DeleteElem (LEL * pdel, LEL * prev)
{
  if (pdel == list) {                    /* видаляється перший елемент */
     list = list-> next;
     free(pdel);
```

```
    return list;              /* вказівник на список після видалення */
  }
  while (prev-> next != pdel)       /* пошук елемента, що передує pdel */
    prev = prev-> next;
  prev->next = pdel-> next;                    /* перемикання зв'язків */
  free (pdel);
  return prev;          /* вказівник на останній перевірений елемент */
}
```

Функція повертає вказівник на новий початок списку, якщо видалено перший елемент, інакше – вказівник на елемент, що передував видаленому. Починаючи з цього елемента, буде виконуватись пошук наступних елементів списку, які підлягають видаленню.

Процес видалення зі списку всіх елементів, що мають непарні індекси, можна реалізувати через циклічні виклики функцій `FindOddIndex()` та `DeleteElem()`. Саме так організовано роботу функції `DeleteAllOdd()`:

```
void  DeleteAllOdd (void)
{
    LEL * pst = list, * pdel;
  while ((pdel = FindOddIndex(pst)) != NULL)      /* поки є елементи з */
    pst = DeleteElem(pdel, pst);  /* непарними індексами - видаляємо */
}
```

За умовами даної конкретної задачі замість трьох функцій, описаних вище, можна записати одну ефективнішу в реалізації функцію `DeleteOddElements()`. У процесі виконання ця функція послідовно перевіряє всі елементи списку та відразу видаляє ті, що відповідають заданій умові. Для перемикання зв'язків у разі видалення елемента використовується додатковий вказівник `prev`. Він зберігає адресу елемента, який передує тому, що видаляється (див. рис. 13.2, *в*).

```
void  DeleteOddElements (void)
{
    LEL * pel = list, * prev;
  while (pel != NULL)
    if (pel-> inform.index%2 == 0 )     /* індекс парний */
      prev = pel;                       /* елемент залишається */
      pel = pel-> next;
    else {                              /* елемент треба видалити */
      if (pel == list) {                /* видалення першого елемента */
        list = list-> next;
        free(pel);
        pel = list;
      } else {                          /* видалення інших елементів */
        prev-> next = pel-> next;       /* перемикання зв'язків */
        free (pel);
        pel = prev-> next;
      }
    }
}
```

Повний текст програми. В поданому далі повному тексті програми використано три окремі функції пошуку та видалення елементів, описані раніше, оскільки вони більш універсальні та гнучкі для доповнень. Ці функції дають змогу змінювати конфігурацію усього однозв'язного списку або опрацьовувати лише окремі його елементи.

```
/*****************************************************************/
/*  Формування однозв'язного динамічного списку-черги з даних,   */
/*  що складаються з текстового повідомлення та його індекса.    */
/*  Видалення зі списку елементів з непарними індексами          */
/*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#define MES_LEN  200                    /* розмірність рядка повідомлення */
typedef struct  inform {                 /* структура інформаційного поля */
          int index;
          char message[MES_LEN];
        } INFORM;
typedef struct  list_elem {                  /* структура елемента списку */
          INFORM inform;
          struct list_elem *next;
} LEL;
LEL *  AddElem (LEL* last);                          /* прототипи функцій */
void  PrintList (void);
LEL *  FindOddIndex (LEL* start);
LEL *  DeleteElem (LEL* pdel, LEL * prev);
void  DeleteAllOdd (void);
void  FreeList (void);
LEL  *list;                     /* глобальний вказівник на початок списку */
int  main (void)
{
    LEL *end = NULL;              /* вказівник на останній елемент списку */
  puts ("\t\t Вхідні дані:");
  do {                                          /* цикл формування списку */
     end = AddElem(end);
  } while (end != NULL);
  puts ("\n\n\t Введено дані:");
  PrintList();                           /* виведення сформованого списку */
  DeleteAllOdd();          /* видалення елементів з непарними індексами */
  puts ("\n\n\t Список після видалення:");
  PrintList();                            /* виведення скороченого списку */
  FreeList();                                          /* звільнення ДП */
  return 0;
}
```

```
/* Функція приєднання до черги нового елемента */
LEL* AddElem (LEL* last)
{
    LEL *pel;                              /* вказівник на новий елемент */
    static int num=1;                      /* номер елемента, що вводиться */
  pel=(LEL*)malloc(sizeof(LEL));           /* виділення ДП для елемента */
  printf("\n\t%d-й елемент:  індекс - ", num);
  scanf("%d", &pel->inform.index);
  if (pel->inform.index==0) {              /* ознака кінця введення */
    free (pel);
    return NULL;
  }
  fflush (stdin);                          /* очищення буфера введення */
  printf("повідомлення:  ");
  gets(pel->inform.message);
  pel->next=NULL;                  /* новий елемент буде останнім у списку */
  if (list==NULL)                          /* якщо список порожній */
    list=pel;
  else
    last->next=pel;                /* приєднання до останнього в списку */
  num++;
  return pel;
}
/* Функція виведення на екран усіх даних списку */
void PrintList (void)
{
    LEL *pel=list;
  while (pel!=NULL) {
    printf("\n%-8d%-70s", pel->inform.index, pel->inform.message);
    pel=pel->next;
  }
}
/* Функція видалення заданого елемента */
LEL* DeleteElem (LEL* pdel, LEL* prev)
{
  if (pdel==list) {                        /* видалення першого елемента */
     list=list->next;
     free(pdel);
     return list;                  /* вказівник на список після видалення */
  }
  while (prev->next!=pdel)           /* пошук елемента, що передує pdel */
      prev=prev->next;
  prev->next=pdel->next;                   /* перемикання зв'язків */
  free(pdel);
  return prev;              /* вказівник на останній перевірений елемент */
}
```

```
/* Функція пошуку елемента з непарним індексом */
LEL * FindOddIndex (LEL* pel)
{
  while (pel!=NULL)
    if ( pel-> inform.index % 2!= 0 )          /* індекс елемента непарний */
      return pel;
    else
      pel = pel-> next;
  return NULL;                          /* не знайдено відповідного елемента */
}
/* Функція видалення всіх елементів з непарними індексами */
void DeleteAllOdd (void)
{
    LEL *pst = list, *pdel;
  while ((pdel = FindOddIndex (pst))!= NULL)       /* циклічний пошук і */
    pst = DeleteElem (pdel, pst);          /* видалення елементів списку */
}
/* Функція витирання всього списку */
void FreeList (void)
{
    LEL *pel = list;
  while (pel!= NULL) {
     list = list-> next;
     free (pel);                          /* витирання поточного елемента */
     pel = list;
  }
}
```

Приклад виконання програми:

```
      Вхідні дані:
    1-й елемент:  індекс - 205
повідомлення:  Понизити рівень сигналу
    2-й елемент:  індекс - 318
повідомлення:  Розпочати процес регулювання
    3-й елемент:  індекс - 202
повідомлення:  Підвищити рівень сигналу
    4-й елемент:  індекс - 765
повідомлення:  Аварійна ситуація
    5-й елемент:  індекс - 381
повідомлення:  Перевірка діапазону частот
    6-й елемент:  індекс - 456
повідомлення:  Завершити контроль
    7-й елемент:  індекс - 0
```

```
      Введено дані:
205     Понизити рівень сигналу
318     Розпочати процес регулювання
202     Підвищити рівень сигналу
765     Аварійна ситуація
381     Перевірка діапазону частот
456     Завершити контроль
      Список після видалення:
318     Розпочати процес регулювання
202     Підвищити рівень сигналу
456     Завершити контроль
```

Програмування стека. Поширеним різновидом однозв'язного списку є стек. Особливість організації стека в тому, що кожен новоприєднаний елемент стає вершиною списку, тому зчитування елементів проводиться в послідовності, зворотній до послідовності їх запису. Зчитаний елемент видаляється зі списку. Як правило, використання стека не передбачає видалення серединних елементів списку.

Програмування стека є частковим випадком роботи з лінійним однозв'язним списком і базується на двох операціях:

- приєднання нового елемента до вершини списку;
- зчитування та видалення першого елемента списку.

Для реалізації цих операцій запишемо дві короткі функції: `PutInStack()` та `GetFromStack()`. Будемо вважати, що елементи стека мають таку саму структуру, що й елементи лінійного списку в попередній програмі. Обидві функції повертають адресу нової вершини списку.

Перша функція `PutInStack()` призначена для створення нового елемента та запису його в стек. Вона використовує один параметр `data`, через який передається вміст інформаційного поля елемента (дані повинні бути введені окремо).

```
/*  Запис нового елемента в стек  */
LEL *  PutInStack (INFORM data)
{
     LEL *pnew;                          /* вказівник на новий елемент */
   pnew = (LEL *)malloc(sizeof(LEL));    /* виділення ДП для елемента  */
   pnew->inform = data;                  /* заповнення поля даних */
   pnew->next = list;                    /* під'єднання до списку */
   list = pnew;                          /* перший у списку */
   return pnew;
}
```

Друга функція `GetFromStack()` зчитує елемент з вершини стека та видаляє його. Перед видаленням функція записує за адресою, заданою вказівником `pdat`, дані інформаційного поля цього елемента.

```
/*  Зчитування елемента зі стека  */
LEL*  GetFromStack (INFORM* pdat)
{
    LEL *ph = list;                          /* вказівник на вершину */
  if (ph == NULL)  return NULL;              /* список порожній  */
  *pdat = ph->inform;                        /* копіювання поля даних */
  list = list->next;                         /* перенесення вершини списку */
  free(ph);                                  /* звільнення ДП */
  return list;
}
```

Використовуючи розроблені функції, можна формувати список-стек, послідовно зчитувати та видаляти його елементи чи доповнювати стек новими даними.

13.3.2. Двозв'язні лінійні списки

Структура елемента двозв'язного списку. Кожен елемент двозв'язного динамічного списку зберігає адреси як наступного, так і попереднього елементів цього списку (див. рис. 13.1,*б*). Це дає змогу рухатись по списку в обох напрямках і підвищує гнучкість та ефективність процесів вставлення та видалення елементів. Правда, процес формування такого списку дещо складніший, ніж однозв'язного.

Проілюструємо роботу з двозв'язним списком на прикладі попередньої задачі, поставивши додаткову умову, що елементи сформованого списку повинні бути впорядкованими за зростанням індексів.

Введемо ще одне доповнення. У попередній програмі інформаційне поле елементів списку формувалось як структура, що складається з двох компонентів: індекса та текстового повідомлення. Розмір повідомлення для всіх елементів був фіксованим. Раніше вже наголошувалось, що такий підхід, з одного боку, обмежує розміри текстових рядків, а з іншого – призводить до нераціональних витрат оперативної пам'яті. Тому в цій програмі зберігатимемо в інформаційному полі індекс елемента та вказівник на рядок текстового повідомлення, а самі повідомлення занесемо в динамічну пам'ять окремо (рис. 13.3). Це дасть змогу виділити для кожного повідомлення необхідну за обсягом ділянку. Відповідно зміняться шаблони структур інформаційного поля та всього елемента списку:

```
typedef struct inform {                  /* шаблон структури     */
          int index;                     /* інформаційного поля */
          char* mes;
        } DMINF;
typedef struct list_elem {               /* шаблон структури */
          DMINF inf;                     /* елемента списку  */
          struct list_elem *next, *prev;
        } EL2WL;
```

тут `next` – вказівник на наступний елемент списку, а `prev` – на попередній.

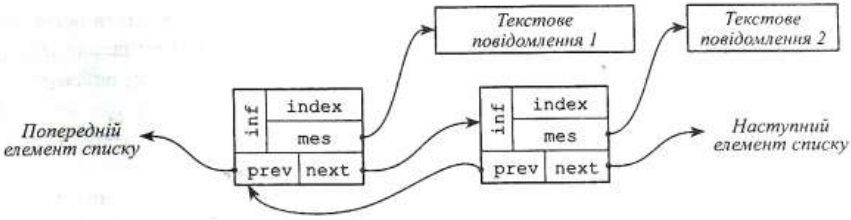


Рис. 13.3. Структура елементів двозв'язного списку з відокремленими текстовими повідомленнями

У програмі використаємо глобальні вказівники `list_beg` та `list_end`, які будуть зберігати адреси відповідно вершини та кінцевого елемента списку:

```
EL2WL *list_beg, *list_end;
```

Формування впорядкованого списку. У циклічному процесі формування впорядкованого двозв'язного лінійного списку можна виділити три основні кроки:

- введення даних і запис рядка повідомлення у динамічну пам'ять;
- визначення місця введених даних у списку відповідно до його впорядкованості;
- вставлення у список нового елемента, сформованого з введених даних.

У програмі ці кроки реалізуємо за допомогою трьох окремих функцій.

Функція `InputData()` виконує введення даних і формує структуру інформаційного поля нового елемента списку. Вона має один параметр `pinf`, що задає адресу структури, в яку заносяться вхідні дані.

```
int  InputData (DMINF * pinf)
{
    char buf[MAX_LEN];            /* буфер введення рядка повідомлення */
    static int num = 0;                                  /* лічильник даних */
  printf ("\n\t%d-й елемент:  індекс - ", ++num );
  scanf ("%d", &pinf->index);
  if (pinf->index!= 0) {          /* кінець введення - нульовий індекс */
     printf ("Повідомлення:  ");
     fflush (stdin);    gets (buf);
     pinf->mes =(char *)malloc(strlen(buf)+1);      /* виділення ДП і */
     strcpy(pinf->mes, buf);                 /* запис рядка повідомлення */
  } else
     pinf->mes = NULL;                            /* введення даних завершено */
  return pinf->index;
}
```

Функція `InputData()` повертає введене значення індекса, яке використовується в `main()` для перевірки, чи процес введення даних завершено (`index` дорівнює `0`). Текстове повідомлення спочатку зчитується в буфер, а потім переноситься у динамічну пам'ять. Адреса повідомлення записується в поле `mes` структури вхідних даних.

Функція `InsertElem()` формує новий елемент у динамічній пам'яті та заповнює його інформаційне поле введеними даними. Якщо список іще порожній, то даний елемент фіксується як перший і останній у списку. За умови непорожнього списку виконується пошук позиції, де має бути вставлений сформований елемент, щоб збереглась загальна впорядкованість списку. Вказівник `pins` встановлюється на перший з елементів списку, індекс якого перевищує індекс нововведеного елемента (рис. 13.4, *б*). Якщо ж індекс нового елемента найбільший, то `pins` набуває значення `NULL`. Коли позицію нового елемента в списку визначено, виконується виклик функції `AddElem()`, яка приєднує сформований елемент перед елементом, на який вказує `pins`, або робить його останнім у списку, якщо `pins` дорівнює `NULL`.

```
void  InsertElem (DMINF data)
{
    EL2WL *pel, *pins;
  pel = (EL2WL *)malloc(sizeof(EL2WL));                    /* виділення ДП */
  pel->inf = data;                        /* заповнення поля даних елемента */
  if (list_beg == NULL) {                              /* якщо список порожній */
     pel->next = pel->prev = NULL;
     list_beg = list_end = pel;                       /* єдиний елемент списку */
     return;
  }
  pins = list_beg;    /* цикл пошуку позиції вставлення нового елемента */
  while (data.index > pins->inf.index && pins != NULL)
     pins = pins->next;
  AddElem (pel, pins);       /* приєднання нового елемента перед *pins */
}
```

Функція `AddElem()` реалізує вставлення нового елемента в список. Вона має два параметри: `pnew` – вказівник на елемент, що вставляється, та `pold` – вказівник на елемент, перед яким треба ввести новий. Можна виділити три випадки:

- новий елемент стає першим у списку;
- новий елемент вводиться у середину списку;
- новий елемент стає останнім у списку.

У першому випадку новий елемент приєднується до вершини списку:

```
list_beg->prev = pnew;
pnew->next = list_beg;
```

Аналогічним чином у третьому випадку новий елемент приєднується до кінця списку:

```
list_end->next = pnew;
pnew->prev = list_end;
```

Другий випадок, коли елемент `*pnew` треба вставити в середину списку, найскладніший, оскільки вимагає перемикання зв'язків двох сусідніх елементів, між якими вводиться новий елемент (рис. 13.4, *б*). Спочатку новостворений елемент приєднується до попереднього та наступного елементів списку:

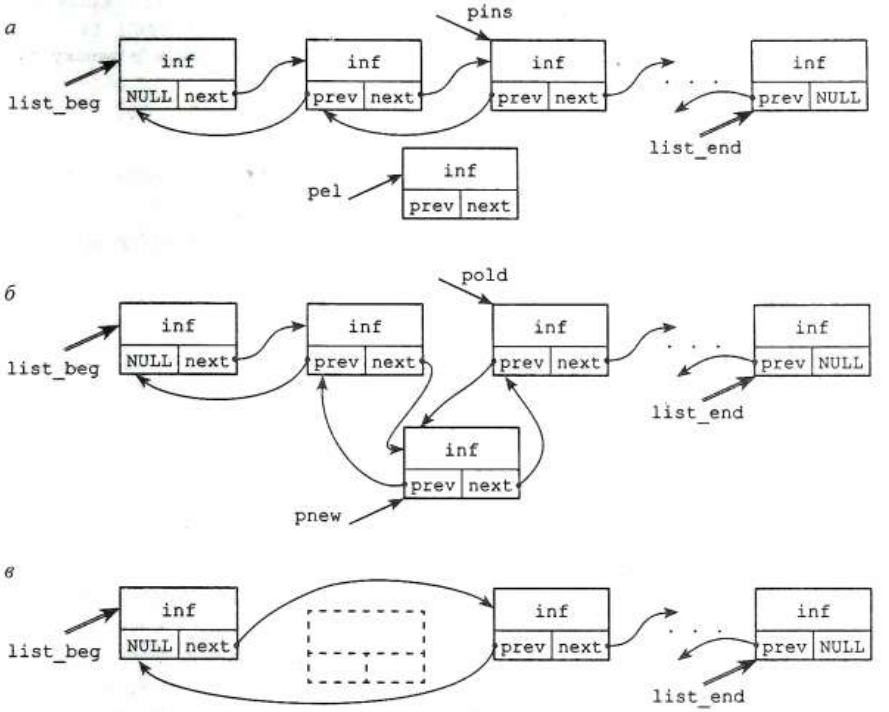


Рис. 13.4. Зміна конфігурації двозв'язного списку:
а – сформований список і новий елемент;
б – під'єднання нового елемента;
в – видалення серединного елемента списку

```
pnew->next = pold;
pnew->prev = pold->prev;
```

З огляду на двозв'язність списку, адресу попередника нового елемента зчитуємо з поля `prev` елемента-наступника `pold->prev`. Відповідне перемикання зв'язків сусідніх елементів, між якими вводиться новий, виконують оператори:

```
pold->prev-> next = pnew;
pold->prev = pnew;
```

Звернемо увагу на два моменти. По-перше, на вираз `pold->prev->next`, який визначає поле `next` елемента, що передує нововведеному (того, з яким раніше був пов'язаний елемент `*pold`). По-друге, на порядок запису операторів перемикання зв'язків – його змінювати не можна, оскільки другий оператор розриває зв'язок елемента `*pold` з попереднім і перемикає його на новий елемент.

```
void AddElem (EL2WL * pnew, EL2WL * pold)
{
  if (pold == list_beg) {       /* новий елемент стає першим у списку */
     list_beg->prev = pnew;
     pnew->next = list_beg;
     pnew->prev = NULL;
     list_beg = pnew;
     return;
  }
  if (pold==NULL) {             /* новий елемент стає останнім у списку */
     list_end->next = pnew;
     pnew->prev = list_end;
     pnew->next = NULL;
     list_end = pnew;
  } else {                 /* новий елемент вводиться в середину списку */
     pnew->next = pold;
     pnew->prev = pold->prev;
     pold->prev->next = pnew;
     pold->prev = pnew;
  }
}
```

Рекурсивне виведення даних списку. Сформований список виведемо на екран двічі. Перший раз скористаємось функцією `PrintList()`, яка друкує дані списку в порядку їх введення (ця функція подібна до функції `PrintList()` з попередньої програми). За другим разом застосуємо функцію `PrintReverse()`, яка друкує список у зворотному порядку: від кінцевого елемента до вершини. На прикладі цієї функції покажемо, що для програмування динамічних списків доцільно використовувати рекурсивні функції, оскільки самі динамічні інформаційні структури рекурсивні за своєю організацією.

```
void PrintReverse (EL2WL * last)
{
  if (last == NULL)  return;                          /* список порожній */
  printf ("\n%-8d%-70s", last->inf.index, last->inf.mes);
  PrintReverse(last->prev);       /* рекурсивне продовження виведення */
}
```

Нагадаємо, що за швидкодією рекурсивні функції здебільшого будуть поступатись звичайним, а в разі дуже довгих списків рекурсивні виклики можуть спричинити переповнення пам'яті стека.

Видалення заданих елементів і всього списку. Видалення всіх елементів списку, які мають непарні індекси, виконує функція `DeleteAllOdd()`. Вона викликає функцію `DeleteElem()`, призначену для видалення заданого елемента списку.

```
void DeleteAllOdd (void)
{
     EL2WL * pel = list_beg;
```

```
  while ( pel!=NULL )
    if ( pel->inf.index%2 != 0 )                    /* індекс непарний */
       pel = DeleteElem(pel);                       /* видалення елемента */
    else
       pel = pel->next;
}
```

Дії, що виконуються у процесі видалення елемента двозв'язного списку, залежать від позиції цього елемента: перший, серединний чи останній елемент. Якщо видаляється перший елемент, то вказівник вершини списку переноситься на наступний елемент. У разі видалення останнього елемента треба перенести вказівник кінця списку на передостанній елемент і зробити цей елемент кінцевим. Якщо ж видаляється елемент з середньої частини списку, то необхідно перемкнути відповідні зв'язки (рис. 13.4,*в*):

```
pdel->next->prev = pdel->prev;
pdel->prev->next = pdel->next;
```

Уважно проаналізуйте конструкцію `pdel->next->prev`, через яку реалізовано звертання до поля `prev` елемента, наступного за тим, що видаляється (тобто наступного за `*pdel`), і аналогічну конструкцію `pdel->prev->next`, яка виконує звертання до поля `next` елемента, що передує `*pdel`.

Функція `DeleteElem()` повертає вказівник на елемент, який перед видаленням був у списку наступним за тим, що видаляється. У випадку видалення серединного елемента адреса наступника попередньо заноситься в окремий вказівник `pnext`.

```
EL2WL *  DeleteElem (EL2WL * pdel)
{
  if (pdel == list_beg) {          /* видалення першого елемента списку */
     list_beg = list_beg->next;
     list_beg->prev = NULL;
     FreeElemMemo(pdel);
     return list_beg;
  }
  if (pdel == list_end) {      /* видалення останнього елемента списку */
     list_end = list_end->prev;
     list_end->next = NULL;
     FreeElemMemo(pdel);
     return NULL;
  } else {                    /* видалення серединного елемента списку */
       EL2WL *pnext;                              /* допоміжний вказівник */
     pdel->next->prev = pdel->prev;
     pnext = pdel->prev->next = pdel->next;
     FreeElemMemo(pdel);
     return pnext;
  }
}
```

Для звільнення динамічної пам'яті, яку займає елемент списку (окремий рядок текстового повідомлення і сама структура елемента списку), у програмі використано допоміжну функцію `FreeElemMemo()`:

```
void  FreeElemMemo (EL2WL * pel)
{
  free(pel->inf.mes);              /* витирання рядка повідомлення */
  free(pel);                       /* витирання елемента списку    */
}
```

Витирання всього списку виконує рекурсивна функція `FreeList()`, яка теж звертається до функції `FreeElemMemo`:

```
void  FreeList (EL2WL * start)
{
   if (start == NULL)              /* список порожній */
     return;
   list_beg = start->next;
   FreeElemMemo(start);            /* витирання початкового елемента */
   FreeList(list_beg);             /* рекурсивне продовження */
}
```

Повний текст програми. Наведемо повний текст програми опрацювання двозв'язного динамічного списку згідно з умовами поставленої задачі.

```
/*****************************************************************/
/*  Створення двозв'язного динамічного списку, впорядкованого    */
/*  за зростанням індексів текстових повідомлень. Видалення зі   */
/*  списку всіх елементів з непарними індексами                  */
/*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LEN  200               /* максимальний розмір повідомлення */
typedef struct  inform {           /* структура інформаційного поля    */
          int index;
          char *mes;               /* вказівник на рядок повідомлення  */
        } DMINF;
typedef struct  list_elem {              /* структура елемента списку */
          DMINF inf;
          struct list_elem *next, *prev;
        } EL2WL;
int InputData (DMINF * inf);             /* прототипи функцій */
void InsertElem (DMINF data);
void AddElem (EL2WL * pnew, EL2WL * pold);
void PrintList (void);
void PrintReverse (EL2WL * last);
```

```
EL2WL * DeleteElem (EL2WL * pdel);
void FreeElemMemo (EL2WL * pel);
void DeleteAllOdd (void);
void FreeList (EL2WL * start);
EL2WL * list_beg, * list_end;      /* вказівники початку й кінця списку */
int  main (void)
{
     DMINF input;
  puts ("\n\t\t Вхідні дані:");
  while (InputData(&input)!= 0)              /* цикл формування списку */
     InsertElem(input);
  puts ("\n\n\t Сформований список:");
  PrintList();                          /* виведення сформованого списку */
  puts("\n\n\t Зворотний порядок даних:");
  PrintReverse(list_end);     /* виведення списку в зворотному порядку */
  DeleteAllOdd();          /* видалення елементів з непарними індексами */
  puts("\n\n\t Список після видалення:");
  PrintList();                          /* виведення скороченого списку */
  FreeList(list_beg);                                  /* звільнення ДП */
  return 0;
}
/* Функція введення даних елемента списку */
int  InputData (DMINF * pinf)
{
     char buf[MAX_LEN];
     static int num = 0;
  printf ("\n\t%d-й елемент:  індекс - ", ++num);
  scanf ("%d", &pinf->index);
  if (pinf->index!= 0) {          /* кінець введення - нульовий індекс */
     printf ("Повідомлення:  ");
     fflush (stdin);    gets (buf);
     pinf->mes =(char*)malloc(strlen(buf)+1);       /* виділення ДП і */
     strcpy(pinf->mes, buf);              /* запис рядка повідомлення */
  } else
     pinf->mes = NULL;
  return pinf->index;
}
/* Функція приєднання нового елемента в порядку зростання індексів */
void  InsertElem (DMINF data)
{
     EL2WL *pel, *pins;
  pel = (EL2WL *)malloc(sizeof(EL2WL));                  /* виділення ДП */
  pel->inf = data;                              /* заповнення поля даних */
```

```
  if (list_beg == NULL) {          /* якщо в списку ще не було елементів */
     pel->next = pel->prev = NULL;
     list_beg = list_end = pel;                  /* єдиний елемент списку */
     return;
  }
  pins = list_beg;     /* цикл пошуку позиції вставлення нового елемента */
  while (data.index > pins->inf.index && pins!= NULL)
     pins = pins->next;
  AddElem(pel, pins);          /* приєднання нового елемента перед *pins */
}

/* Функція приєднання до списку елемента *pnew перед елементом *pold */
void  AddElem (EL2WL * pnew, EL2WL * pold)
{
  if (pold == list_beg) {          /* новий елемент стає першим у списку */
     list_beg->prev = pnew;
     pnew->next = list_beg;
     pnew->prev = NULL;
     list_beg = pnew;
     return;
  }
  if (pold!=NULL) {        /* новий елемент вводиться у середину списку */
     pnew->next = pold;
     pnew->prev = pold->prev;
     pold->prev->next = pnew;
     pold->prev = pnew;
  } else {                        /* новий елемент стає останнім у списку */
     list_end->next = pnew;
     pnew-> prev = list_end;
     pnew-> next = NULL;
     list_end = pnew;
  }
}

/* Функція виведення даних списку в прямому порядку */
void  PrintList (void)
{
    EL2WL * pel = list_beg;
  while (pel != NULL) {
     printf ("\n%-8d%-70s", pel->inf.index, pel->inf.mes);
     pel = pel->next;
  }
}

/* Рекурсивна функція виведення даних у зворотному порядку */
void  PrintReverse (EL2WL * last)
{
  if (last==NULL)  return;
```

```
  printf ("\n%-8d%-70s", last->inf.index, last->inf.mes);
  PrintReverse(last->prev);                    /* рекурсивне продовження */
}

/* Функція видалення елемента списку (повертає вказівник на наступний
   за видаленим елемент) */
EL2WL * DeleteElem (EL2WL * pdel)
{
  if (pdel == list_beg) {          /* видалення першого елемента списку */
     list_beg = list_beg->next;
     list_beg->prev = NULL;
     FreeElemMemo(pdel);
     return list_beg;
  }
  if (pdel == list_end) {        /* видалення останнього елемента списку */
     list_end = list_end->prev;
     list_end->next = NULL;
     FreeElemMemo(pdel);
     return NULL;
  } else {                      /* видалення серединного елемента списку */
       EL2WL *pnext;
     pdel->next->prev = pdel->prev;
     pnext=pdel->prev->next = pdel->next;
     FreeElemMemo(pdel);
     return pnext;
  }
}

/* Функція видалення всіх елементів списку з непарними індексами */
void DeleteAllOdd (void)
{
    EL2WL * pel = list_beg;
  while (pel!= NULL)
     if (pel->inf.index%2 != 0)
        pel = DeleteElem(pel);                    /* видалення елемента */
     else
        pel = pel->next;
}

/* Рекурсивна функція витирання списку */
void FreeList (EL2WL * start)
{
   if (start == NULL)  return;
   list_beg = start->next;
   FreeElemMemo(start);                /* витирання початкового елемента */
   FreeList(list_beg);                          /* рекурсивне продовження */
}
```

```
/* Функція звільнення ДП, зайнятої елементом списку */
void  FreeElemMemo (EL2WL * pel)
{
  free(pel->inf.mes);                    /* витирання рядка повідомлення */
  free(pel);                             /* витирання елемента списку    */
}
```

У разі виконання програми для тих самих вхідних даних, що використовувались для тестування однозв'язних списків (див. параграф 13.3.1), отримуємо результат:

```
      Сформований список:
202      Підвищити рівень сигналу
205      Понизити рівень сигналу
318      Розпочати процес регулювання
381      Перевірка діапазону частот
456      Завершити контроль
765      Аварійна ситуація

      Зворотний порядок даних:
765      Аварійна ситуація
456      Завершити контроль
381      Перевірка діапазону частот
318      Розпочати процес регулювання
205      Понизити рівень сигналу
202      Підвищити рівень сигналу

      Список після видалення:
202      Підвищити рівень сигналу
318      Розпочати процес регулювання
456      Завершити контроль
```

13.4. Двійкові дерева

Базові означення. На відміну від списків, які базуються на лінійній організації зв'язків між елементами, *дерева* є розгалуженими ієрархічно організованими динамічними інформаційними структурами. У практичних задачах, пов'язаних з інформаційними технологіями, застосовують різні типи дерев, проте найпоширенішими є двійкові дерева, які дають змогу здійснювати швидкий напрямлений пошук даних.

Для опису дерев прийнято певну термінологію. Елемент дерева називають *вузлом* або *вершиною*. Найстарший вузол, що є вершиною всього дерева, називають *коренем* (рис. 13.5). Корінь дерева розташований на найвищому першому рівні. Вузол дерева може мати нащадків, такий вузол називають *батьківським*, а вузли-нащадки – *дочірніми* вузлами. Кожен дочірній вузол пов'язаний з одним батьківським, а рівень дочірніх вузлів на один нижчий, ніж у батьківського вузла (рис. 13.6). Кількість дочірніх елементів визначає *степінь* вузла, а найбільший допустимий для вузлів степінь

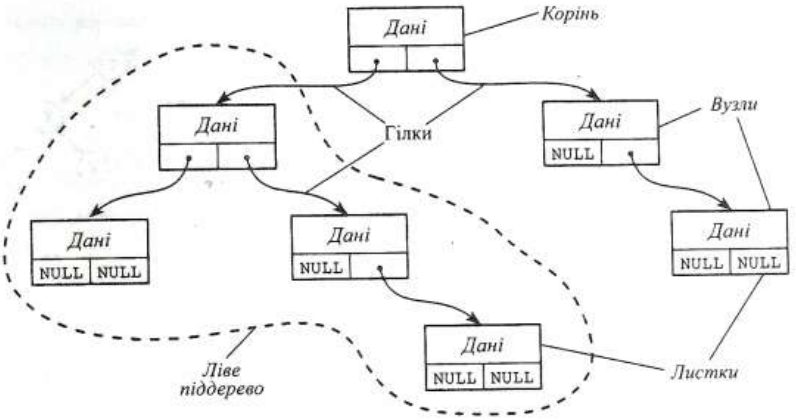


Рис. 13.5. Структура двійкового дерева

називають *степенем дерева*. Елементи дерева, які не мають дочірніх вузлів, називають *листками* або *термінальними* (кінцевими) елементами. Частину дерева, що зберігає деревоподібну структуру, називають *піддеревом*.

Двійковими (або *бінарними*) називають дерева, кожен елемент яких складається з інформаційної частини та двох вказівників: на лівий і на правий дочірній вузли. Степінь усіх вузлів двійкового дерева не перевищує двох. Окремим видом двійкового дерева є *дерево пошуку* – впорядковане двійкове дерево, в якому кожен лівий дочірній вузол містить дані, ключовий елемент яких менший, ніж у батьківського вузла, а кожен правий дочірній вузол – дані, ключовий елемент яких більший за ключ батьківського вузла (*ключ* – певна ознака, за якою здійснюється впорядкування). Таким чином, у двійковому дереві пошуку кожне ліве піддерево включає вузли, ключові елементи яких менші за ключ кореневого елемента, а кожне праве піддерево складається з вузлів, значення ключів яких більші, ніж ключ кореня (рис. 13.6). Надалі будемо вести мову саме про двійкові дерева пошуку, хоча для скорочення вживатимемо загальний термін дерево.

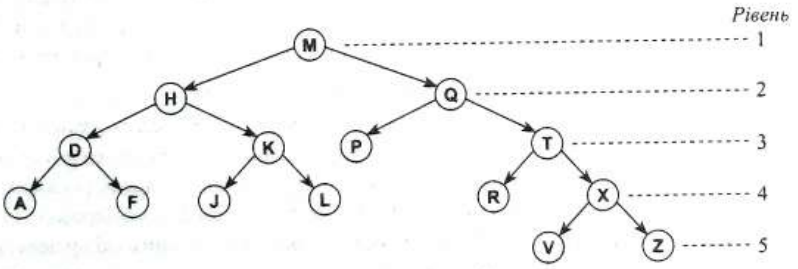


Рис. 13.6. Приклад двійкового дерева пошуку, впорядкованого за ключами-літерами

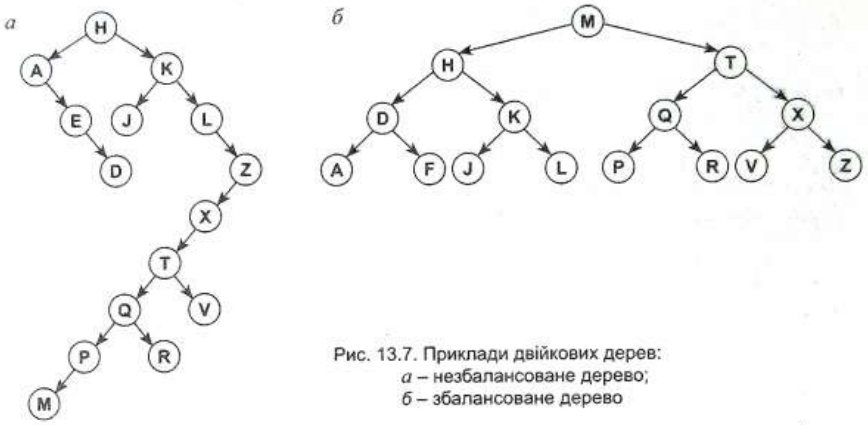


Рис. 13.7. Приклади двійкових дерев:
а – незбалансоване дерево;
б – збалансоване дерево

Висотою дерева називають максимальну кількість рівнів від кореня до кінцевого елемента (листка). Приміром, висота дерева, зображеного на рис. 13.6, дорівнює 5. Слід зауважити, що впорядковане дерево з тими самими вершинами може мати різну конфігурацію і висоту. На рис. 13.7 зображено два варіанти дерев з такими ж вузлами, як і в дерева з рис. 13.6, перше з яких має висоту 9, а друге – 4. З рис. 13.7,*а* видно, що дерево може виродитись у лінійний список, тоді його висота дорівнюватиме загальній кількості вузлів дерева. Тому важливе значення має *збалансованість* дерева. Дерево вважається збалансованим, якщо коефіцієнт розбалансованості для кожного його піддерева потрапляє в межі від -1 до 1. *Коефіцієнтом розбалансованості* називають різницю висот лівого і правого піддерев. Так, дерева на рис. 13.6 і 13.7,*а* не збалансовані, а дерево, зображене на рис. 13.7,*б*, – збалансоване. Відразу зазначимо, що процес збалансування, який необхідно виконувати після кожної зміни конфігурації дерева, достатньо складний і нами розглядатись не буде. Тих, хто захоче глибше познайомитись з питаннями організації і програмування деревоподібних структур, скеровуємо до [4, 12, 23].

Пошук даних у двійковому дереві базується на його впорядкованості. Щоб знайти вузол, ключовий елемент якого збігається зі заданим, дерево переглядають від кореня, просуваючись тільки по тих гілках, в яких може бути необхідний вузол. В найгіршому випадку кількість вершин, перевірених у процесі пошуку, дорівнюватиме висоті дерева. Для повністю збалансованого дерева пошуку, що має n вузлів, це значення становить $[\log_2 n]+1$, де [ ] позначають цілу частину виразу.

Формування двійкового дерева. Основні прийоми роботи з двійковими деревами пошуку розглянемо на прикладі такої задачі. З клавіатури вводиться послідовність англійських слів-термінів і їх перекладів українською мовою. Із введених даних треба сформувати двійкове дерево-словник. Визначити висоту створеного дерева та відобразити його на екрані. Потім організувати пошук перекладів заданих слів у дереві-словнику. Передбачити можливість видалення окремих вузлів і доповнення дерева новими вузлами. Наприкінці роботи реалізувати витирання всього дерева.

Вузол дерева оголосимо як структуру, подібну до елемента двозв'язного списку:

```
typedef struct tree_node {
          WORDS wrd;                              /* поле даних */
          struct tree_node *left, *right;         /* вказівники на */
        } TreeNode;                               /* дочірні елементи */
```

Інформаційне поле вузла `wrd` є вкладеною структурою, яка містить два вказівники на символьні рядки, в яких записано слово-термін і його переклад:

```
typedef struct words {
          char *eng;                              /* слово-термін в ДП */
          char *ukr;                              /* переклад слова */
        } WORDS;
```

Самі ж символьні рядки з даними будемо зберігати в динамічній пам'яті окремо.

Алгоритм формування двійкового дерева подібний до алгоритму формування динамічного списку і включає три основні кроки:

- введення даних інформаційного поля;
- формування вузла дерева;
- впорядковане приєднання нового вузла до дерева.

Перший крок: введення термінів, їх перекладів і запис введених даних у динамічну пам'ять – виконує функція `GetData()`, текст якої подано в повній програмі опрацювання двійкового дерева.

Новий вузол дерева формує функція `NewNode()`, яка виділяє динамічну пам'ять для цього вузла і заносить туди адреси текстових рядків, введених через виклик функції `GetData()`. Перед завершенням функція `NewNode()` повертає адресу сформованого вузла.

```
TreeNode *  NewNode (void)
{
    TreeNode *pel;
    WORDS buf;
  if (GetData(&buf)== 0)                          /* кінець введення */
    return NULL;
  pel=(TreeNode *)malloc(sizeof(TreeNode));
  pel->wrd = buf;                          /* заповнення нового вузла */
  pel->left = pel->right = NULL;
  return pel;
}
```

Перший сформований вузол стає коренем дерева, його адреса зберігатиметься в глобальному вказівнику `root`. Другий вузол приєднується до кореня як його лівий або правий нащадок з урахуванням вимог впорядкованості. Всі наступні вузли приєднуються відповідно до лівого або правого піддерева вже сформованого дерева. Кожен новий вузол формується як листок дерева і приєднується до батьківського вузла так, щоб загальне дерево було впорядкованим у словниковій формі за словами-термінами.

Приєднання до дерева нового сформованого вузла виконує рекурсивна функція `AddNode()`, яка має два параметри: `pnew` – вказівник на вузол, який треба приєднати

до дерева як листок, і root_adr – адресу вказівника на корінь дерева (піддерева), до якого приєднується новий вузол. Коли буде знайдено місце приєднання новоствореного вузла, ця адреса визначатиме вказівник у батьківському вузлі, який пов'язується з приєднаним елементом. Пошук місця приєднання нового вузла розпочинається з кореня дерева. Якщо дерево порожнє, то новий елемент приймається за корінь дерева. В іншому разі процес пошуку рекурсивно продовжується у лівому або правому піддереві. Коли ж виявиться, що слово-термін уже наявне в словнику, то процес приєднання вузла припиняється і звільняється динамічна пам'ять, виділена для збереження даних нового вузла.

```
void AddNode (TreeNode * pnew, TreeNode ** root_adr)
{
    TreeNode *proot = *root_adr;          /* вказівник на корінь */
    int cmp;
  if (proot == NULL) {                    /* дерево порожнє */
     *root_adr = pnew;            /* новий елемент стає коренем дерева */
     return;
  }
  cmp = strcmp(proot->wrd.eng, pnew->wrd.eng);
  if (!cmp)                               /* слово вже є    */
     FreeNodeMemo(pnew);                  /* звільнення ДП */
  else
     if (cmp > 0)
       AddNode(pnew, &proot->left);       /* ввести у ліве піддерево  */
     else
       AddNode(pnew, &proot->right);      /* ввести у праве піддерево */
}
```

Ми не виконуємо збалансування сформованого дерева, оскільки цей процес достатньо трудомісткий і алгоритмічно непростий. Будемо опиратися на ту властивість, що в разі випадковості вхідних даних та їх значної кількості, отримане дерево наближається до збалансованого [12].

Функція AddNode() рекурсивна, як і більшість інших функцій цієї програми. Нерекурсивний варіант функції буде помітно довшим і менш наочним. Застосування рекурсії для опрацювання дерев пов'язане передусім з тим, що за своєю організацією дерево є рекурсивною інформаційною структурою. Цю властивість дерев добре ілюструє лаконічна функція обчислення висоти дерева TreeHeight(). Висота кожного непорожнього дерева (піддерева) визначається через висоту більшого з його піддерев.

```
int TreeHeight (TreeNode * proot)
{
    int lh,rh;
  if (proot == NULL) return 0;            /* дерево порожнє */
  lh = TreeHeight(proot->left);           /* висота лівого піддерева  */
  rh = TreeHeight(proot->right);          /* висота правого піддерева */
  return lh > rh ? lh+1 : rh+1;           /* +1 - враховує корінь */
}
```

Спробуйте написати нерекурсивний варіант функції `TreeHeight()`, і відразу постане питання, як переходити з одного піддерева на інше, щоб не допустити повторень і не пропустити жодного вузла.

Обхід дерева. Ряд задач, наприклад, виведення на екран усіх даних дерева чи витирання дерева, вимагають проходження по всіх його вузлах. У цих випадках важливим питанням є порядок обходу дерева. Використовують три варіанти обходу дерев:

- *симетричний* (або *зліва направо*), коли спочатку обходять ліве піддерево, потім корінь, а за ним – праве піддерево;
- *прямий* (або *зверху вниз*), при якому першим опрацьовується корінь, а потім за чергою ліве та праве піддерева;
- *нижній* (або *знизу вверх*), коли спочатку опрацьовується ліве піддерево, потім праве, а в останню чергу – корінь.

У разі застосування симетричного обходу порядок проходження вузлів для всіх трьох дерев, зображених на рис. 13.6 і рис. 13.7, буде однаковим:

A - D - F - H - J - K - L - M - P - Q - R - T - V - X - Z

Для двох інших способів обходу послідовність опрацювання вузлів залежить від конфігурації дерева. Наприклад, для дерева з рис. 13.6 прямий порядок обходу відповідає такій послідовності вузлів:

M - H - D - A - F - K - J - L - Q - P - T - R - X - V –Z

У разі обходу цього дерева знизу послідовність звертання до вершин буде такою:

A - F - D - J - L - K - H - P - R - V - Z - X - T - Q - M

Наведена нижче функція `PrintTree()` здійснює симетричний обхід вузлів дерева і виводить на екран слова-терміни та їх переклади.

```
void PrintTree (TreeNode * proot)
{
  if (proot == NULL)  return;              /* дерево порожнє */
  PrintTree (proot->left);                 /* обхід лівого піддерева */
 /* виведення даних кореневого елемента */
  printf ("%-20s -  %s\n", proot->wrd.eng, proot-> wrd.ukr);
  PrintTree (proot->right);                /* обхід правого піддерева */
}
```

Нижній обхід дерева виконує коротка рекурсивна функція `FreeTree()`, яка звільняє усю динамічну пам'ять, зайняту деревом.

```
void FreeTree (TreeNode * proot)
{
  if (proot == NULL)  return;
  FreeTree (proot->left;);          /* витирання лівого піддерева */
  FreeTree (proot->right);          /* витирання правого піддерева */
  FreeNodeMemo (proot);             /* видалення кореневого елемента */
}
```

Наприкінці розділу наведено функцію `DeleteAllNodes()`, яка реалізує ітераційний спосіб обходу всіх вузлів дерева в прямому порядку.

Для відображення ієрархічної структури дерева запишемо функцію `ShowTree()`, яка виводить ключові терміни всіх вузлів, розташовуючи кожен на відповідному рівні. Щоб забезпечити можливість відтворення дерев різної висоти й конфігурації, виведене дерево буде повернутим на 90° проти годинникової стрілки. Тому обхід дерева у функції виконується справа наліво (в зворотному до симетричного порядку). Слово-термін кожного вузла виводиться в окремому рядку і зсувається від початку рядка на `lev*8` позицій, де `lev` – рівень даного вузла.

```
void  ShowTree (TreeNode * proot, int lev)
{
  if (proot == NULL)  return;
  ShowTree (proot->right, lev+1);   /* відображення правого піддерева */
  printf ("%*c%s\n", lev*8, ' ', proot->wrd.eng);  /* термін кореня */
  ShowTree (proot->left, lev+1);     /* відображення лівого піддерева */
}
```

Пошук даних у дереві. Основним призначенням двійкового дерева є швидкий пошук даних. Як вже зазначалось, у впорядкованому дереві процес пошуку ефективний, оскільки перевіряються не всі вузли, а тільки ті, що згідно з впорядкованістю дерева можуть містити шукані дані. Функція `FindNode()` визначає вузол, в якому зберігається адреса заданого терміна `term`, і повертає вказівник на цей вузол або `NULL`, якщо такого терміна в словнику немає. З огляду на лінійність процесу просування по дереву використаємо ітераційний алгоритм, щоб забезпечити вищу швидкодію процесу пошуку.

```
TreeNode *  FindNode (char * term)              /* пошук слова term */
{
    TreeNode *pnode = root;
    int cmp;
  while (pnode!= NULL) {                        /* цикл пошуку слова   */
    cmp = strcmp(pnode->wrd.eng, term);         /* порівняння термінів */
    if (cmp == 0)  return pnode;                /* слово знайдено */
    else  if (cmp > 0)                      /* продовження пошуку слова  */
          pnode = pnode->left;              /* шукати в лівому піддереві */
       else
          pnode = pnode->right;             /* шукати в правому піддереві */
  }
  return NULL;                              /* слово не знайдено */
}
```

Видалення вузла дерева. Дії, що виконуються у разі видалення вузла дерева, залежать від того, скільки дочірніх елементів має даний вузол. Якщо вузол є листком дерева, то, щоб видалити його, треба звільнити пам'ять від усіх даних цього вузла, а в адресне поле батьківського елемента занести `NULL`. Якщо вузол, що видаляється, має тільки один дочірній елемент, то цей елемент треба перенести на місце видаленого, що підтягне на один рівень угору всі вузли його піддерева (рис. 13.8,*б*).

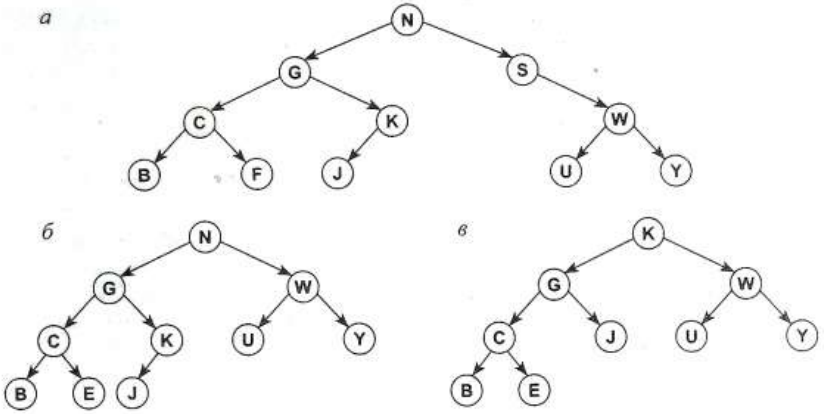


Рис. 13.8. Видалення вузлів двійкового дерева: *а* – початкове дерево; *б* – видалення вузла S, який мав один дочірній елемент; *в* – видалення вузла N, який мав два дочірні елементи

Найскладніший випадок, коли видаляється вузол, що має обидва піддерева – з них треба утворити спільне впорядковане дерево. На місце видаленого вузла можна перенести його лівий або правий дочірній елемент. Але така зміна конфігурації може збільшити висоту дерева, оскільки одне з піддерев нової вершини буде приєднане до другого піддерева. Щоб максимально зберегти структуру та впорядкованість дерева, застосуємо інший прийом: на місце видаленого вузла перенесемо крайній правий вузол лівого піддерева (рис. 13.8,*в*). Ключ цього вузла (K) найбільший у лівому піддереві, але менший від усіх ключів правого піддерева. Тому за значеннями ключів цей вузол найближчий до кореня (N), який видаляється. Так само можна використати крайній лівий вузол правого піддерева (U). На місце вузла заміни (K) піднімаємо його лівий дочірній елемент (J) і приєднуємо його до батьківського вузла (G).

Рекурсивна функція пошуку й видалення вузла дерева `DeleteNode()` має два параметри: перший – `term` задає слово-термін, яке визначає вузол, що підлягає видаленню, а другий – `root_adr` задає адресу вказівника на корінь дерева, в якому має виконуватись пошук вузла. За цією адресою функція запише адресу нового кореня дерева. Видалення вузла, слово-термін якого збігається з `term`, реалізовано за наступним алгоритмом. Визначаємо кількість піддерев цього вузла (для наочності використано відповідні макроконстанти). У разі відсутності піддерев або наявності тільки одного з них, відповідно змінюємо значення вказівника за адресою `root_adr`. Якщо ж вузол має обидва піддерева, то, рухаючись по правій гілці лівого піддерева, знаходимо вузол, що не має правого нащадка. Цей вузол замінить видалений. Приєднуємо до нього праве піддерево видаленого вузла. Якщо вузол заміни не є вершиною лівого піддерева видаленого вузла, то пов'язуємо його батьківський вузол з його лівим дочірнім вузлом. Потім приєднуємо до вузла заміни все ліве піддерево. Після перебудови дерева витираємо старий вузол та його дані з динамічної пам'яті, використовуючи функцію `FreeNodeMemo()`.

```
#define  NoSubTree     0
#define  LeftSubTree  -1
#define  RightSubTree  1
#define  TwoSubTrees   2
void  DeleteNode (char* term, TreeNode** root_adr)
{
    TreeNode *proot = *root_adr, *pnewr, *ppar;
    int cmp, subtr;
  if (proot == NULL)    return;                          /* дерево порожнє */
  cmp = strcmp(proot->wrd.eng, term);                 /* порівняння термінів */
  if (cmp == 0) {                                        /* слово знайдено */
    if (proot->left == NULL && proot->right == NULL)     /* визначення */
      subtr = NoSubTree;                                 /* складу піддерев */
    else if (proot->left == NULL)
           subtr = RightSubTree;
         else if (proot->right == NULL)
                subtr = LeftSubTree;
              else  subtr = TwoSubTrees;
    switch (subtr) {                                  /* перебудова дерева */
      case NoSubTree:                                    /* немає піддерев */
        *root_adr = NULL;   break;
      case LeftSubTree:                          /* є тільки ліве піддерево */
        *root_adr = proot->left;   break;
      case RightSubTree:                        /* є тільки праве піддерево */
        *root_adr = proot->right;  break;
      case TwoSubTrees:                             /* є обидва піддерева */
        pnewr = proot->left;                  /* початок пошуку вузла заміни */
        ppar = proot;                          /* і його батьківського вузла */
        while (pnewr->right != NULL) { /* просування по правій гілці */
          ppar = pnewr;
          pnewr = pnewr->right;
        }
        pnewr->right = proot->right;  /* приєднання правого піддерева */
        if (pnewr != proot->left) {   /* не вершина лівого піддерева  */
          ppar->right = pnewr->left;  /* перемикання зв'язків *pnewr  */
          pnewr->left = proot->left;  /* приєднання лівого піддерева  */
        }
        *root_adr = pnewr;                          /* зміна кореня */
    }
    FreeNodeMemo(proot);        /* видалення старого кореневого вузла */
    return;
  }
  if (cmp > 0)                    /* видалення вузла в лівому піддереві */
     DeleteNode (term, &proot->left);
  else                           /* видалення вузла в правому піддереві */
     DeleteNode (term, &proot->right);
}
```

Щоб продемонструвати, як видалення вузлів змінило загальну конфігурацію дерева, у програмі після видалення слів зі словника повторно викликається функція `ShowTree()`. Для більшої наочності структури дерева, перед відображенням вузлів виводиться рядок розмітки рівнів дерева. Цю дію виконує функція `ShowLevels()`, яка звертається до функції `TreeHeight()` для визначення висоти дерева.

Повний текст програми опрацювання двійкового дерева. Наведемо повний текст програми, яка реалізує формування, відображення, зміну конфігурації та витирання двійкового дерева пошуку.

```
/*********************************************************************/
/*  Формування двійкового дерева, інформаційними елементами якого є  */
/*  слова-терміни та їх переклади. Відображення структури дерева.    */
/*  Пошук перекладів заданих слів. Видалення окремих вузлів дерева   */
/*********************************************************************/

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

typedef struct words {
          char *eng *ukr;      /* вказівники на рядки, записані в ДП */
} WORDS;
typedef struct tree_node {
          WORDS wrd;
          struct tree_node, *left, *right;
} TreeNode;                          /* шаблон структури вузла дерева */

int  GetData (WORDS * pwrd);                        /* прототипи функцій */
TreeNode *  NewNode (void);
void  AddNode (TreeNode * pnew, TreeNode ** root_adr);
void  PrintTree (TreeNode * proot);
void  ShowTree (TreeNode * proot, int lev);
int  TreeHeight(TreeNode * proot);
TreeNode *  FindNode (char * term);
void  DeleteNode (char * term, TreeNode ** root_adr);
void  FreeNodeMemo (TreeNode * pnode);
void  FreeTree (TreeNode * proot);
void  ShowLevels (void);

TreeNode * root;                          /* вказівник на корінь дерева */

int  main (void)
{
    TreeNode * node;
    char term[30];

  puts("\t Формування дерева:");
  while ((node = NewNode()) != NULL)          /* цикл формування дерева */
     AddNode(node, &root);
  puts("\n\t Сформоване дерево:\n");
  PrintTree(root);                              /* відображення дерева */
```

```
  printf("\nВисота дерева - %d:  лівого піддерева - %d,  "
         "правого піддерева - %d\n", TreeHeight(root),
         TreeHeight(root->left), TreeHeight(root->right));
  puts("\n\t  Структура дерева:");
  ShowTree(root, 1);
  printf("\n\t Пошук термінів: \nТермін -> ");
  while(*gets(term)!= '-') {                              /* цикл пошуку */
     if ((node=FindNode(term))!= NULL)              /* слово є в дереві */
        printf("   переклад:  %s\n", node->wrd.ukr);
     else  puts(" Такого терміна немає");
     printf("Наступний термін -> ");
  }
  printf("\n\t Видалення слів: \n Термін -> ");
  while(*gets(term)!= '-')  {                          /* цикл видалення */
     DeleteNode(term, &root);
     printf(" Наступний термін -> ");
  }
  puts("\nСтруктура дерева після видалення:");
  ShowTree(root, 1);                  /* відображення зміненого дерева */
  FreeTree(root);  root=NULL;                          /* витирання дерева */
  return 0;
}

/* Функція введення слів-термінів і їх перекладів */
int  GetData (WORDS * pwrd)
{
    static int num = 1;
    char wbuf[150];
  printf("%d-й термін ->  ", num++);
  if (*gets(wbuf)== '-')                          /* ознака кінця введення */
     return 0;
  pwrd->eng = (char*)malloc(strlen(wbuf)+1);
  strcpy(pwrd->eng, wbuf);                      /* запис слова-терміна в ДП */
  printf("\tпереклад ->  ");
  gets(wbuf);
  pwrd->ukr =(char *)malloc(strlen(wbuf)+1);
  strcpy(pwrd->ukr, wbuf);                        /* запис перекладу в ДП */
  return 1;
}

/* Функція звільнення ДП, зайнятої вузлом дерева */
void  FreeNodeMemo (TreeNode * pnode)
{
  free(pnode->wrd.eng);                          /* витирання рядків у ДП */
  free(pnode->wrd.ukr);
  free(pnode);                                      /* витирання вузла */
}
```

```
/* Функція формування нового вузла дерева */
TreeNode * NewNode (void)
{
     TreeNode *pel;
     WORDS buf;
   if (GetData(&buf)== 0)                          /* кінець введення */
      return NULL;
   pel=(TreeNode *)malloc(sizeof(TreeNode));
   pel-> wrd = buf;                              /* заповнення нового вузла */
   pel->left = pel-> right = NULL;           /* новий елемент стає листком */
   return pel;
}

/* Функція приєднання нового вузла до впорядкованого дерева */
void AddNode (TreeNode * pnew, TreeNode ** root_adr)
{
    TreeNode *proot = *root_adr;                  /* вказівник на корінь */
    int cmp;
   if (proot == NULL) {                        /* дерево порожнє */
     *root_adr = pnew;              /* новий елемент стає коренем дерева */
     return;
  }
  if (!(cmp = strcmp(proot->wrd.eng, pnew->wrd.eng)))    /* слово вже є */
     FreeNodeMemo(pnew);                                  /* звільнення ДП */
  else  if (cmp > 0)
       AddNode(pnew, &proot->left);          /* ввести у ліве піддерево */
     else
       AddNode(pnew, &proot->right);        /* ввести у праве піддерево */
}

/* Функція виведення даних дерева в симетричному порядку */
void PrintTree (TreeNode * proot)
{
  if (proot == NULL)   return;
  PrintTree(proot->left);                    /* виведення лівого піддерева */
  printf("%-15s - %s\n", proot->wrd.eng, proot-> wrd.ukr); /* корінь */
  PrintTree (proot->right);                 /* виведення правого піддерева */
}

/* Функція відображення структури дерева */
void ShowTree (TreeNode * proot, int lev)
{
  if (lev == 1)  ShowLevels();        /* виведення рядка розмітки рівнів */
  if (proot == NULL)  return;
  ShowTree (proot->right, lev+1);     /* відображення правого піддерева */
  printf ("%*c %s\n", lev*8, ' ', proot->wrd.eng); /* виведення кореня */
  ShowTree (proot->left, lev+1);       /* відображення лівого піддерева */
}
```

```
/* Функція обчислення висоти дерева */
int  TreeHeight (TreeNode * proot)
{
    int lh,rh;
  if (proot == NULL) return 0;
  lh = TreeHeight(proot->left);            /* висота лівого піддерева  */
  rh = TreeHeight(proot->right);           /* висота правого піддерева */
  return lh > rh ? lh+1 : rh+1;                /* +1 - враховує корінь */
}

/* Функція виведення рядка розмітки рівнів */
void  ShowLevels (void)
{
    int lvl;
  printf("Рівень:  ");
  for (lvl = 1;  lvl <= TreeHeight(root);  lvl++)
     printf("%-8d", lvl);
  printf("\n");
}

/* Функція пошуку заданого слова в бінарному дереві */
TreeNode *  FindNode (char* term)
{
    TreeNode * pnode = root;
    int cmp;
  while (pnode!= NULL) {                       /* цикл пошуку слова   */
     cmp = strcmp(pnode->wrd.eng, term);       /* порівняння термінів */
     if (cmp == 0)  return pnode;                  /* слово знайдено */
     else
        if (cmp > 0)
           pnode = pnode->left;           /* пошук у лівому піддереві  */
        else
           pnode = pnode->right;          /* пошук у правому піддереві */
  }
  return NULL;                                    /* слово не знайдено */
}

/* Функція видалення вузла дерева, що містить задане слово */
#define  NoSubTree     0
#define  LeftSubTree  -1
#define  RightSubTree  1
#define  TwoSubTrees   2
void  DeleteNode (char * term, TreeNode ** root_adr)
{
     TreeNode * proot = *root_adr, * pnewr, * ppar;
     int cmp, subtr;
  if (proot == NULL)    return;
```

```
cmp = strcmp(proot->wrd.eng, term);              /* порівняння термінів */
if (cmp == 0)  {                          /* видалення кореневого елемента */
   if (proot->left == NULL && proot-> right == NULL)   /* визначення */
      subtr = NoSubTree;                                /* складу піддерев */
      else if (proot->left == NULL)
              subtr = RightSubTree;
           else if (proot->right == NULL)
                   subtr = LeftSubTree;
                else
                   subtr = TwoSubTrees;
   switch (subtr) {                                    /* перебудова дерева */
      case NoSubTree:                                    /* немає піддерев */
        *root_adr = NULL;    break;
      case LeftSubTree:                          /* є тільки ліве піддерево */
        *root_adr = proot->left;    break;
      case RightSubTree:                        /* є тільки праве піддерево */
        *root_adr = proot->right;  break;
      case TwoSubTrees:                                  /* є обидва піддерева */
        pnewr = proot->left;                        /* *pnewr - новий корінь */
        ppar = proot;                            /* батьківський вузол *pnewr */
        while (pnewr->right != NULL) {              /* пошук вузла заміни */
          ppar = pnewr;
          pnewr = pnewr->right;
        }
        pnewr->right = proot->right;  /* приєднання правого піддерева */
        if (ppar != proot) {
          ppar->right = pnewr->left;              /* перемикання зв'язків */
          pnewr->left = proot->left;                /* у лівому піддереві */
        }
        *root_adr = pnewr;                                  /* зміна кореня */
    }
    FreeNodeMemo(proot);          /* витирання старого кореневого вузла */
    return;
  }
  if (cmp > 0)                          /* видалення вузла в лівому піддереві */
     DeleteNode (term, &proot->left);
  else                                 /* видалення вузла в правому піддереві */
     DeleteNode (term, &proot-> right);
}

/* Функція повного витирання дерева */
void  FreeTree (TreeNode * proot)
{
  if (proot == NULL)   return;                              /* дерево порожнє */
  FreeTree(proot->left);                          /* витирання лівого піддерева  */
  FreeTree (proot->right);                        /* витирання правого піддерева  */
  FreeNodeMemo(proot);                            /* видалення кореневого елемента */
}
```

Приклад виконання програми:

```
      Формування дерева:
1-й термін ->  media
    переклад ->  носій даних, інформації
2-й термін ->  key
    переклад ->  клавіша; ключ; шифр
3-й термін ->  scale
    переклад ->  масштаб; шкала
4-й термін ->  template
    переклад ->  шаблон
5-й термін ->  linker
    переклад ->  компонувальник, редактор зв'язків
6-й термін ->  backup
    переклад ->  резервна копія
7-й термін ->  password
    переклад ->  пароль
8-й термін ->  unit
    переклад ->  пристрій; елемент
9-й термін ->  -
       Сформоване дерево:
backup          -  резервна копія
key             -  клавіша; ключ; шифр
media           -  носій даних, інформації
linker          -  компонувальник, редактор зв'язків
password        -  пароль
scale           -  масштаб; шкала
template        -  шаблон
unit            -  пристрій; елемент
Висота дерева - 4:  лівого піддерева - 2,  правого піддерева - 3
           Структура дерева:
Рівень:   1       2       3       4
                                  unit
                          template
                  scale
                          password
          media
                          linker
                  key
                          backup
      Пошук термінів:
  Термін ->  scale
    переклад:  масштаб; шкала
  Термін ->  backup
    переклад:  резервна копія
```

```
Термін ->  capability
  переклад:   Такого терміна немає
Термін ->  -

      Видалення слів:
Термін ->  template
Наступний термін ->  media
Наступний термін ->  -

Структура дерева після видалення:

Рівень:  1      2      3
                       unit
                scale
                       password
        linker
                key
                       backup
```

Нерекурсивний обхід дерева. На завершення розглянемо нерекурсивний варіант функції витирання дерева. Видалення всіх елементів – це одна із задач, що вимагає повного обходу дерева. У функції `DeleteAllNodes()` реалізовано метод прямого обходу дерева – зверху вниз. Щоб зафіксувати не пройдені ще вузли і піддерева, у функції використано стек, в який заносяться адреси правих дочірніх елементів вузлів, що видаляються. Просування вниз здійснюється по лівій гілці піддерева. Коли витерто термінальний елемент або все ліве піддерево, зі стека зчитується адреса вузла, яку було занесено останньою, і процес повторюється для наступного піддерева. Витирання дерева завершується, коли пройдено всі вузли і стек порожній.

```
/* Ітераційний варіант функції витирання дерева */

void DeleteAllNodes (void)
{
    TreeNode *pdel = root, *pnext;      /* починаємо з кореня */
  while (pdel!= NULL) {
     if (pdel->left!= NULL) {
        pnext = pdel->left;             /* просування по лівій гілці */
        if (pdel->right!= NULL)                 /* є праве піддерево */
           ToStack(pdel->right);    /* запис адреси піддерева в стек */
     } else
        if (pdel->right!= NULL)
           pnext = pdel->right;         /* перехід на праве піддерево */
        else
           pnext = FromStack();   /* зчитування адреси вузла зі стека */
     FreeNodeMemo(pdel);                          /* видалення вузла */
     pdel = pnext;                          /* перехід на наступний вузол */
   }
}
```

Стек, в якому зберігаються адреси вершин невитертих піддерев, можна організувати як статичний масив (обмеження накладає апріорна невизначеність розмірності) або як однозв'язний динамічний список (до програми доведеться додати функції роботи зі списком, див. параграф 13.3.1). Ще один спосіб – використання динамічного масиву, розмір якого дорівнює висоті дерева, що має бути витертим:

```
    TreeNode ** stack;                  /* вказівник на початок стека */
stack=(TreeNode **)calloc(TreeHeight(root), sizeof(TreeNode *));
```

Виділення динамічної пам'яті для елементів стека, які є адресами вузлів дерева, виконано функцією `calloc()`, що відразу онулює виділену область. Якщо перший елемент масиву-стека не змінювати, то він буде `NULL`-вказівником. Зчитування `NULL` можна використовувати як ознаку завершення процесу витирання дерева. Запис адрес вузлів у такий масив-стек і зчитування їх зі стека можна виконувати операторами:

```
stack[++i] = pdel->right;           /* замість  ToStack(pdel->right); */
pnext = stack[i--];                 /* замість  pnext = FromStack(); */
```

тут `i` – лічильник адрес вузлів, занесених у стек, початково `i` дорівнює 0.

Запитання та завдання для самоконтролю

1. Яку частину пам'яті називають динамічною? Чи має вона іншу назву? Які переваги розташування даних у динамічній пам'яті?
2. Назвіть стандартні бібліотечні функції, призначені для звертання до динамічної пам'яті. Яку дію виконує кожна з них?
3. Використовуючи функцію `malloc()`, виділіть у динамічній пам'яті місце для символьного рядка, розмір якого може сягати 500 символів, а за допомогою функції `calloc()` – ділянку для масиву, що складається з 500 довгих цілих чисел. У чому відмінності результатів виконання цих двох функцій?
4. Задекларуйте структурний тип `PHONE_CALL`, призначений для збереження інформації про міжміський телефонний дзвінок: *<Дата>*, *<Час>*, *<Місто>*, *<Номер_телефона>*, *<Тривалість_розмови>*. Оголосіть масив `mcalls` із 120-и вказівників на структури `PHONE_CALL` і виділіть динамічну пам'ять для кожної структури. Припустимо, що всі структури вже заповнені даними. Як перевірити, чи перша цифра номера телефона, на який було зроблено *k*-й дзвінок, є цифрою 4?
5. Як створити масив вказівників для попередньої задачі, якщо розмірність масиву `mcalls` стає відомою тільки в процесі виконання програми? Створіть відповідний масив вказівників. Як тепер найшвидше переставити місцями дві задані структури з введеного набору даних? А як видалити з цього набору всі дані про дзвінки в м. Харків?
6. З чого складається елемент однозв'язного динамічного списку? Як формується такий список? Чим відрізняються списки-стеки від списків-черг? Як видалити з однозв'язного списку заданий елемент?
7. Для задачі з параграфа 13.3.1 напишіть рекурсивну функцію, призначену для витирання початкової частини списку: від елемента вершини списку до заданого елемента.

8. Які переваги двозв'язних списків, коли доцільно їх створювати? Які дії треба виконати, щоб ввести додатковий елемент у середину списку? Які зв'язки перемикають, якщо видаляється серединний елемент двозв'язного списку?
9. Як організовані двійкові дерева? Яка структура вузла дерева? Що називають коренем дерева? Які вузли є листками дерева? Як визначається висота дерева?
10. Які двійкові дерева вважаються впорядкованими? Що таке збалансованість дерева? На що вона впливає? Чи залежить час пошуку в двійковому дереві від його збалансованості?
11. Які способи застосовують для повного обходу дерева? Як змінити функцію `PrintTree()` з параграфа 13.4, щоб виведення даних двійкового дерева здійснювалось у прямому порядку (зверху вниз)? Як реалізувати обхід дерева у нижньому порядку (знизу вверх)?
12. Напишіть функцію, призначену для видалення кореня дерева, що має два піддерева. Новим коренем має стати вершина лівого піддерева, а функція повинна повернути адресу цієї вершини. Як перемкнути праве піддерево вершини, що стала коренем?

Запрограмуйте наведені нижче задачі, використовуючи для збереження даних динамічну пам'ять

13. Сформувати динамічний масив з `N` випадкових трицифрових цілих чисел. Відсортувати елементи масиву в порядку зростання їх значень. Вивести на екран відсортований масив і визначити, скільки в ньому елементів, значення яких більше за значення `LIM`. Звільнити пам'ять, зайняту масивом.
14. Створити динамічний масив, що буде складатись із десяти вказівників, призначених для збереження початків символьних рядків. Занести в динамічну пам'ять введені з клавіатури символьні рядки, виділивши для кожного з них ділянку потрібного обсягу (кінець введення – порожній рядок). Якщо виявиться, що рядків більше, ніж десять, то розмір масиву вказівників збільшити вдвічі за допомогою функції `realloc()`. Витерти ті рядки, які містять задане ключове слово. Надрукувати рядки, що залишились, і звільнити всю виділену динамічну пам'ять.
15. Задано певний вираз, у якому використовуються всі три види дужок: `()`, `[]` і `{}`. Перевірити, чи дужки у виразі розставлені правильно. Для цього треба посимвольно переглянути вираз: якщо зустрілась ліва дужка, то занести її в стек, якщо ж знайдено праву дужку, то перевірити, чи вона парна до дужки, занесеної в стек останньою. Вираз вважається правильним, якщо в кінці перевірки стек порожній.
16. З клавіатури вводиться послідовність символьних рядків, кожен з яких є двійковим кодом довгого цілого числа. З введених рядків і їх десяткових значень у динамічній пам'яті сформувати двозв'язний список. Вивести на екран таблицю введених двійкових кодів і їх десяткових значень. Потім із даних сформованого списку утворити новий однозв'язний список, у якому елементи будуть розташовані так, щоб значення чисел формували спадну послідовність. У процесі формування однозв'язного списку попередній список витирається. Вивести відсортовану послідовність даних, після чого витерти новий список. У програмі використати окрему функцію для обчислення десяткового значення числа за його двійковим кодом.

Розділ 14

МОДЕЛІ ПАМ'ЯТІ BORLAND C. КОРОТКІ Й ДОВГІ ВКАЗІВНИКИ

У цьому розділі:

- Принцип сегментної адресації оперативної пам'яті
- Шість моделей розподілу оперативної пам'яті для коду та даних С-програм: Tiny, Small, Medium, Compact, Large, Huge
- Короткі та довгі вказівники; модифікатори вказівників: `near`, `far`, `huge`; спеціальні короткі вказівники: `_cs`, `_ds`, `_ss`, `_es`
- Безпосереднє звертання до відеопам'яті у текстових режимах відображення інформації
- Функції бібліотеки Borland C для звертання до внутрішньої та зовнішньої динамічної пам'яті

Ефективність безпосереднього звертання до даних в оперативній пам'яті пов'язана з урахуванням особливостей апаратно-програмної організації конкретних операційних середовищ. Матеріал цього розділу не належить до стандартних засобів мови С. Моделі пам'яті, короткі та довгі вказівники, додаткові функції для роботи з динамічною пам'яттю – все це розширення стандарту С, реалізовані в системі програмування Borland C. За допомогою цих засобів програміст може керувати розподілом ресурсів оперативної пам'яті та встановлювати способи адресації, які забезпечують швидкий доступ до об'єктів програми.

14.1. Сегментна організація пам'яті

Операційна система MS DOS, апаратна платформа якої базується на архітектурі мікропроцесорів Intel 8086/8088, використовує сегментну адресацію оперативної пам'яті. Пояснимо коротко її суть. Адреса кожного байта пам'яті зберігається у формі двох складових частин: *сегмента* адреси (*segment*) і *зміщення* адреси (*offset*). Обидві складові є 16-розрядними беззнаковими цілими числами. Коли відбувається звертання до даних у пам'яті, з цих двох складових частин формується єдина 20-розрядна фізична адреса. Формування фізичної адреси здійснюється за такою схемою:

1) значення сегмента розширюється до 20 біт і зсувається вліво на 4 розряди; біти, що звільнилися справа, заповнюються нулями;
2) до отриманої 20-розрядної сегментної адреси додається значення зміщення; якщо виникає переповнення, то його ігнорують.

Повна фізична адреса є 20-розрядною, що дає змогу адресувати 2^{20} = 1 Мбайт оперативної пам'яті. Цю пам'ять використовує в звичайному режимі роботи (*real mode*) операційна система MS DOS.

У письмових позначеннях адреси сегмент і зміщення прийнято розділяти двокрапкою – *segment*:*offset*. Якщо вказуються константні значення складових адреси, то їх записують у шістнадцятковій формі з літерою h у кінці (від *heximal* – шістнадцятковий), наприклад: 2C60:00A5h. Одну і ту ж фізичну адресу можна отримати з різних значень *segment*:*offset*. Три наступні пари: 0DD0:0008h, 0D00:0D08h і 0320:AB08h адресують один і той самий байт оперативної пам'яті, фізична адреса якого – 0DD08h.

Сегментну компоненту адреси називають також номером параграфа чи просто *параграфом* адреси, а в мікропроцесорах Intel 8086/8088 параграфом позначають межу пам'яті, кратну 16. Оскільки при формуванні фізичної адреси сегмент зсувається вліво на 4 розряди, то для довільного значення *segment*:0000h відповідна фізична адреса буде містити нулі в останній тетраді, тобто буде кратною 16. До сегмента додається зміщення, яке є 16-розрядним словом, тому в межах одного сегмента можна адресувати 2^{16} = 65566 = 64 Кбайт оперативної пам'яті.

Для розташування *exe*-коду С-програми та даних, які вона опрацьовує, виділяються окремі сегменти оперативної пам'яті. Кожен сегмент починається з адреси, кратної 16, і має обсяг, що не перевищує 64 Кбайт. Повну фізичну адресу даних і кодів команд програми мікропроцесор формує зі значення сегментної компоненти та значення зміщення. Для збереження сегментних компонентів адрес використовують чотири системних регістри мікропроцесора, які називають сегментними: CS, DS, SS і ES. Призначення їх таке:

- регістр CS (*code segment* – сегмент коду) задає сегментну частину адрес команд програми; область, яку він адресує, називається сегментом коду програми;
- регістр DS (*data segment* – сегмент даних) задає сегментну частину адрес глобальних і статичних змінних програми; область, яку він адресує, називається сегментом даних;
- регістр SS (*stack segment* – сегмент стека) разом з регістром SP (SS:SP) адресує область пам'яті, яка називається програмним стеком і використовується для збереження локальних даних функцій. Переважно заповнення стека виконується від його вершини (найбільшої адреси) вниз, тому значення SP зменшується після кожного запису даних у стек. Коли ж поточна функція завершує роботу, область її даних звільняється, а значення SP збільшується на відповідну величину;
- регістр ES (*extra segment* – додатковий сегментний регістр) використовують, коли потрібно окремо задавати сегментну частину адреси даних (це називають *переадресацією* сегмента), а також для швидкого звертання до оперативної пам'яті, наприклад, у разі копіювання даних з однієї ділянки пам'яті в іншу.

Якщо всі глобальні та статичні дані програми можна розмістити в одному сегменті, а всі команди вміщаються в один сегмент коду, то в процесі виконання такої програми значення сегментних регістрів не змінюються, а для адресації даних і команд достатньо зберігати тільки зміщення адрес. Такий розподіл оперативної пам'яті забезпечує компактність команд і високу швидкодію їх виконання. Проте обсяг сегмента обмежений значенням 64 Кбайт, тому для задач з великими обсягами даних і/або об'ємними кодами програм потрібні багатосегментні моделі пам'яті.

14.2. Моделі розподілу пам'яті

Середовище Borland C дає користувачеві змогу встановити модель розподілу пам'яті, яка найкраще відповідає обсягам даних і коду програми та забезпечує найбільш раціональні форми адресації даних і команд. Тим самим підтримується висока ефективність процесу виконання програми.

Модель пам'яті є одним із параметрів компіляції, які керують виділенням ресурсів для етапу реалізації програми. Вибір моделі пам'яті здійснюють через пункт головного меню інтегрованого середовища Options/Compiler/Code generation/Model. Можна вибрати один з шести альтернативних варіантів: Tiny, Small, Medium, Compact, Large і Huge. Модель пам'яті задає розподіл сегментних блоків оперативної пам'яті для коду і даних програми, а також визначає тип вказівників, обсяг стека програми та вид динамічної пам'яті. Основні характеристики всіх моделей пам'яті зведено в табл. 14.1, де вказано, яким є розподіл оперативної пам'яті для коду програми (код), глобальних і статичних даних (дані), локальних даних (стек). Також зазначено вид динамічної пам'яті (ДП), яку можна використовувати в програмі, та типи вказівників, що будуть встановлені за замовчуванням.

Таблиця 14.1

Характеристики моделей пам'яті Borland C

Найменування моделі			Розподіл оперативної пам'яті				Тип вказівників	
			код	дані	стек	ДП	дані	функції
Tiny	мінімальна	молодші	один спільний сегмент			внутрішня	near	near
Small	мала		1 сегмент	один спільний сегмент		внутр./зовн.	near	near
Medium	середня		сегм./файл	один спільний сегмент		внутр./зовн.	near	far
Compact	компактна	старші	1 сегмент	1 сегмент	1 сегмент	зовнішня	far	near
Large	велика		сегм./файл	1 сегмент	1 сегмент	зовнішня	far	far
Huge	гігантська		сегм./файл	сегм./файл	1 сегмент	зовнішня	far	far

Найменша модель пам'яті Tiny має обмежене застосування, оскільки для коду програми та всіх її даних, включаючи динамічні, виділяється тільки один сегмент оперативної пам'яті. Для невеликих за обсягом програм найчастіше використовують

модель Small, яка передбачає виділення окремого сегмента для коду програми та другого сегмента для збереження глобальних і локальних даних програми (рис. 14.1). Ця модель дає змогу програмувати два види динамічної пам'яті: *внутрішню* (її ще називають *ближньою*), яка використовує незайняту частину сегмента даних, та *зовнішню* динамічну пам'ять (її називають *дальньою*), яка займає вільну частину оперативної пам'яті в межах початкових 640 Кбайт. Модель Medium подібна до Small за розподілом пам'яті для даних, але вона розрахована на багатофайлові програми великого обсягу, тому для коду кожного програмного файла виділяється окремий сегмент. Три згадані моделі пам'яті називають молодшими.

Рис. 14.1. Схема розподілу оперативної пам'яті в моделі Small

Старші моделі: Compact, Large і Huge використовують окремий сегмент для стека та лише зовнішню динамічну пам'ять. Для великих багатофайлових програм доцільно застосовувати розподіл пам'яті за моделлю Large, яка зберігає код кожного програмного файла в окремому сегменті (рис. 14.2). Якщо ж обсяг статичних даних програми перевищує 64 Кбайт, то слід встановити модель Huge. Детальнішу інформацію про особливості розподілу оперативної пам'яті для кожної з моделей можна знайти в [8].

14.3. Короткі та довгі вказівники. Модифікатори вказівників

Молодші моделі пам'яті використовують один спільний сегмент для всіх даних програми. Вони передбачають, що сегментна частина адрес усіх даних зберігається в системному регістрі DS. Тому для адресації змінних достатньо задавати тільки двобайтові значення зміщень. Відповідні вказівники, які зберігають зміщення фізичних

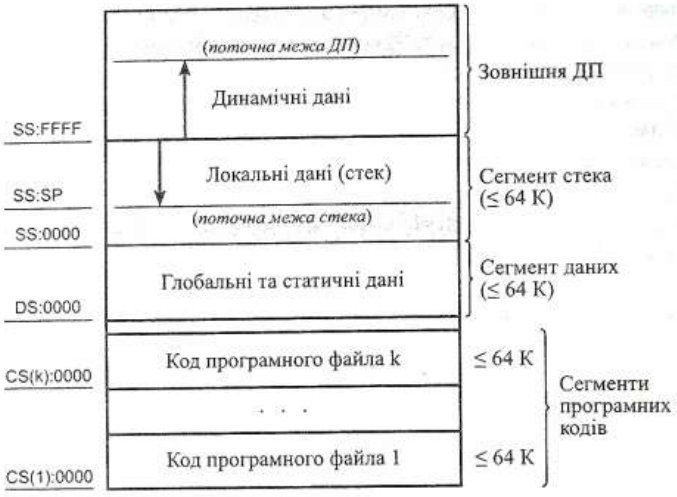


Рис. 14.2. Схема розподілу оперативної пам'яті в моделі Large

адрес, називають *короткими* (чи *ближніми*) або `near`-вказівниками. За замовчуванням двобайтовий формат встановлено для всіх вказівників у молодших моделях пам'яті (див. табл. 14.1).

У старших моделях пам'яті для даних і коду програми виділяється декілька сегментів оперативної пам'яті та використовується зовнішня динамічна пам'ять. Тому в цих моделях адреса зберігається в чотирибайтовому форматі: два старші байти задають сегмент, а два молодші – зміщення адреси. Вказівники старших моделей називають *довгими* (чи *дальніми*) або `far`-вказівниками. Коли відбувається звертання до об'єкта через `far`-вказівник, сегмент адреси записується у відповідний сегментний регістр для формування фізичної адреси цього об'єкта. Довгі вказівники дають змогу звернутись до кожного байта оперативної пам'яті в межах перших 1 Мбайт.

Формат вказівника, встановлений автоматично за заданою моделлю пам'яті, можна змінити, використовуючи модифікатори `near`, `far` або `huge` (всі три є зарезервованими словами Borland C). Модифікатор формату записують після базового типу вказівника перед символом `*`, наприклад:

```
char far* psymb;
```

`near`-, `far`- та `huge`-вказівники. Вказівники з модифікатором `near` є 16-розрядними беззнаковими цілими числами, що зберігають тільки зміщення адреси. Їх доцільно застосовувати в програмах, сумарний обсяг даних яких не перевищує обсягу сегмента (64 Кбайт). Застосування коротких вказівників у багатосегментних програмах вимагає перевизначення сегментного регістра DS або використання спеціальних форм `near`-вказівників, про які мова йтиме далі.

Вказівники з модифікатором far 32-розрядні, вони зберігають повну адресу об'єкта, тобто сегмент і зміщення. Такі вказівники застосовують у молодших моделях пам'яті, коли потрібно звернутись до об'єктів, розташованих за межами сегмента коду програми чи сегмента даних. Зокрема, far-вказівники використовують для роботи з даними у зовнішній динамічній пам'яті, для безпосереднього звертання до відеопам'яті, для доступу до адресного простору BIOS тощо.

Наведемо приклад функції, яка дає змогу записати заданий символ в останню позицію екрана (правий нижній кут). Нагадаємо, що в разі заповнення цієї позиції будь-якою стандартною функцією виведення даних, відбувається автоматичний скролінг екранного зображення – зсунення усіх рядків на один угору. Тому функція заповнення останнього знакомісця має важливе практичне значення. Детальніше питання програмування відеопам'яті в текстових режимах роботи відеосистеми розглянемо в наступному параграфі та в розділі 16.

```
/*  Функція виведення символа в позиції екрана (80,25)  */
void  PutLastScrSmb (char symb, char attr)
{
   /* symb - код символа; attr - атрибути: колір символа і колір фону */
    char far* VMadr = (char far*)0xb8000000; /* початок відеопам'яті */
  VMadr += (24*80+79)*2;          /* адреса останнього символа екрана */
  *VMadr = symb;                                /* відображення символа */
  if (attr)                                  /* якщо треба змінити кольори */
     *(VMadr+1) = attr;
}
```

У разі роботи з довгими вказівниками часто виникає потреба поділу значення адреси на складові частини: сегмент і зміщення, чи, навпаки, необхідність формування довгого вказівника зі складових частин. Для виконання цих операцій призначені спеціальні макроси, оголошені в заголовному файлі <dos.h>:

```
unsigned  FP_SEG (void far* ptr);
unsigned  FP_OFF (void far* ptr);
void far*  MK_FP (unsigned seg, unsigned off);
```

Два перших макроси FP_SEG() і FP_OFF() виділяють з адреси, занесеної у far-вказівник ptr, відповідно сегментну частину та зміщення. Макрос MK_FP() формує значення довгого вказівника зі заданого сегмента seg і зміщення off. За допомогою цього макроса адресу останнього символа екранного зображення, яка використовувалась у функції PutLastScrSmb(), можна сформувати, наприклад, так:

```
char  far* VMadr = (char far*)MK_FP(0xb800, 25*80*2-2);
```

Якщо тепер оголосити:

```
unsigned  ofs = FP_OFF(VMadr);          /* виділення зміщення */
```

то змінна ofs отримає значення 3998, що відповідає зміщенню в байтах (відносно початку відеопам'яті) знакомісця екрана з координатами (80, 25).

Іншим способом формування довгих адрес та виділення їх складових частин є використання структур і об'єднань. Такий підхід дає змогу одночасно виконувати перетворення базового типу вказівника, тобто змінювати форму доступу до даних.

Розглянемо два наступних оголошення:

```
struct mem_addr {
   unsigned ofs, segm;        /* зміщення та сегмент фізичної адреси */
};
union addr_convert {
   struct mem_addr adr;
   char far *pb;                    /* вказівники   */
   unsigned far *pw;                /* на дані      */
   unsigned long far *pl;           /* різних типів */
} uptc;
```

Змінна `uptc` має тип об'єднання `union addr_convert`, до складу якого входять структура `adr` і три вказівники з різними базовими типами: `pb`, `pw` і `pl`. Поля `ofs` і `segm` структури `adr` задають зміщення і сегмент потрібної фізичної адреси.

Оскільки в `far`-вказівниках сегментна компонента займає два старші байти, а зміщення – два молодші, то поля структури `adr` треба записувати у відповідному порядку: спочатку зміщення, а потім сегмент адреси.

Зміна значень полів структури `adr` викликає відповідну зміну значення кожного з вказівників, оголошених у полях об'єднання `uptc`. Наприклад, наступні присвоєння:

```
uptc.adr.segm = 0x0;                    /* сегмент адреси */
uptc.adr.ofs = 0x0450;                  /* зміщення адреси */
```

сформують адресу 0000:0450h, починаючи з якої в оперативній пам'яті зберігається інформація BIOS про поточні координати текстового курсора для кожної з восьми сторінок відеопам'яті. Тепер до цих BIOS-даних можна звертатись побайтово, використовуючи вказівник `pb`, двобайтовими словами через вказівник `pw` або чотирибайтовими словами через вказівник `pl`. Зокрема, поточну позицію курсора на нульовій (основній) відеосторінці можна отримати так:

```
cur_x = *uptc.pb;               /* горизонтальна координата */
cur_y = *(uptc.pb+1);           /* вертикальна координата */
```

Або можна зчитати відразу обидві координати:

```
cur_pos = *uptc.pw;             /* читання двобайтового слова */
```

Використання `far`-вказівників пов'язане з певними проблемами, про які необхідно пам'ятати, розробляючи програми з цими вказівниками. По-перше, одну і ту ж фізичну адресу можна задати різними значеннями вказівника. Обидва оголошені вказівники:

```
unsigned far* adr1 = (unsigned far*)0x00000410;
unsigned far* adr2 = (unsigned far*)0x00410000;
```

задають спільну адресу початку області даних відеопараметрів BIOS. Проте в разі порівняння цих вказівників: `adr1==adr2` – результат буде хибним, бо порівнюються числові значення, записані в `adr1` та `adr2`. А в разі порівняння `far`-вказівників операціями <, <=, >, >= перевіряються тільки зміщення, тобто молодші половини адрес. Другою проблемою є міжсегментний перехід. Якщо до адреси `far`-вказівника додати певне число, то змінюється тільки зміщення адреси, а перенесення в сегментну половину не відбувається.

Вказаних недоліків не мають вказівники з модифікатором `huge`. В операціях над цими вказівниками беруть участь усі 32 розряди їхніх значень. Після кожної операції виконується нормалізація вказівника: значення сегмента коректується так, щоб значення зміщення не перевищувало 15. Наприклад, адреса 8C65:12A4h у нормалізованій формі записується так: 8D8F:0004h. Нормалізація забезпечує однозначність запису кожної фізичної адреси й коректність усіх операцій порівняння та адресної арифметики, в т. ч. пов'язаних із міжсегментними переходами. Тому для безпомилкової роботи з блоками даних, обсяги яких перевищують обсяг сегмента, треба використовувати `huge`-вказівники, хоча при цьому дещо збільшується час виконання програми через додаткові операції нормалізації.

Наступна функція застосовує `huge`-вказівник для визначення суми всіх машинних слів адресного простору BIOS, починаючи з адреси F000:0000h до FFFF:000Fh включно. Ця сума є унікальною для кожної серії персональних комп'ютерів і може бути використана в програмах захисту інформації.

```
/*  Функція обчислення суми слів BIOS-області. Варіант 1  */
unsigned long  Summa_BIOS (void)
{
  unsigned long sum=0;
  unsigned huge* ph=(unsigned huge*)0xf0000000; /* початок області */
    while (ph)              /* остання адреса - ffff:000fh */
       sum += *ph++;
    return sum;
}
```

`_cs`-, `_ds`-, `_ss`- та `_es`-вказівники. Ефективність безпосереднього програмування оперативної пам'яті зростає у разі застосування спеціальних коротких вказівників, що оголошуються через модифікатори `_cs`, `_ds`, `_ss` та `_es` (ці модифікатори теж є службовими словами Borland C). Короткі вказівники зберігають тільки зміщення фізичних адрес, а сегментна частина адреси береться з відповідного системного регістра: CS, DS, SS чи ES. Для запису в сегментні регістри необхідної адреси початку сегмента використовують спеціальні псевдорегістрові змінні: `_CS`, `_DS`, `_SS` та `_ES`. Нижче подано альтернативний варіант функції обчислення суми машинних слів адресного простору BIOS, в якому для звертання до даних використано спеціальний короткий вказівник `_es`. Час виконання даного варіанта функції істотно менший, ніж попереднього, де використовувався `huge`-вказівник.

```
/*  Функція обчислення суми слів BIOS-області. Варіант 2  */
unsigned long  BIOS_summa (void)
{
  unsigned _es* p_es = (unsigned _es*)0;       /* початкове зміщення */
  unsigned long sum;
   _ES = 0xf000;                          /* сегмент адреси BIOS-області */
   for (sum = *p_es++;  *p_es;  p_es++)
     sum += *p_es;                         /* сума даних усього сегмента */
   return sum;
}
```

Застосування спеціальних near-вказівників та відповідних псевдорегістрових змінних вимагає особливої обережності, оскільки в процесі виконання програми значення сегментних регістрів можуть змінюватись. У цьому випадку, щоб уникнути формування помилкових фізичних адрес, перед наступним звертанням до даних через near-вказівники з модифікаторами _cs, _ds, _es або _ss необхідно відновити потрібне значення відповідного сегментного регістра.

У наступному параграфі наведено ще два приклади застосування _es-вказівників у функціях зміни та копіювання текстових відеозображень.

14.4. Безпосереднє програмування відеопам'яті

Безпосереднє звертання до відеопам'яті комп'ютера є не тільки найшвидшим способом виведення інформації на екран у текстових режимах роботи відеосистеми, а й надає програмістові додаткові можливості для формування та опрацювання текстових повідомлень. Зокрема, саме таким способом (або через BIOS-переривання) можна:

- зчитати символ і/або його атрибути зі заданої позиції екрана;
- змінити виведені на екран символи і/або їх атрибути;
- вивести символ у правій нижній позиції екрана або текстового вікна, уникнувши скролінга;
- записати і/або зчитати текстову інформацію з ненульової сторінки відеопам'яті.

Стандартним текстовим режимом, який автоматично встановлюється для відеоадаптерів VGA та SVGA, є режим номер 3 (25 рядків по 80 символів, 16-кольорів). Вся інформація, що виводиться на екран, зберігається у спеціальній області оперативної пам'яті, яка називається відеопам'яттю. Кожен символ займає у відеопам'яті два байти: у молодшому байті зберігається код, а в старшому – атрибути даного символа (колір

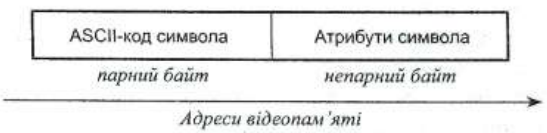


Рис. 14.3. Розташування символів у відеопам'яті

фону, колір і яскравість символа, прапорець блимання). Байт коду символа завжди записується у відеопам'ять за парною адресою, а байт атрибутів – за непарною (рис. 14.3). Структура байта атрибутів описана в параграфі 16.1.1 та проілюстрована на рис. 16.1.

Відеопам'ять текстових режимів входить у загальний адресний простір центрального мікропроцесора. Початок відеопам'яті для всіх кольорових відеоадаптерів має однакову адресу B800:0000h, а для монохромних відеоадаптерів – B000:0000h. Всю відеопам'ять текстових режимів поділено на 8 сторінок, які пронумеровано від 0 до 7. У режимі 25×80 символів обсяг відеосторінки становить 4 Кбайт (4096 байтів). У кожен момент часу на екрані може відображатись тільки одна сторінка, стандартно активною є нульова сторінка відеопам'яті.

Оскільки сторінки відеопам'яті мають фіксовані адреси, то можна безпосередньо звертатись до кожного байта екранного зображення. Зміна значень байтів, що потрапляють у межі активної відеосторінки, відразу ж відображається у відповідній зміні символів і/або атрибутів текстових повідомлень на екрані. Для звертання до відеопам'яті використовують або `far`-вказівники (як це було зроблено в функції `PutLastScrSmb()`, наведеній у попередньому параграфі), або спеціальні короткі `_es`-вказівники (приклади подамо далі). За сегментний компонент адреси здебільшого приймають адресу початку відеопам'яті (B800h для кольорових відеоадаптерів), а зміщення обчислюють за координатами символа на екрані з урахуванням номера відеосторінки:

$$зміщення = ((рядок - 1)\times 80 + (позиція - 1))\times 2 + сторінка \times 4096$$

Першою запишемо функцію, яка повертає код і атрибути символа, що висвітлюється у позиції `col` екранного рядка `row` і зберігається на нульовій сторінці відеопам'яті.

```
/*  Функція зчитування символа зі заданої позиції екрана  */
include <dos.h>
unsigned  GetScrSymbol  (int row, int col)
{
    unsigned  far *addr=(unsigned far*)MK_FP(0xb800, 0);
  addr += (row-1)*80+(col-1);      /* адреса символа у відеопам'яті  */
  return (*addr);                  /* повертає код і атрибути символа */
}
```

Для звертання до двобайтових слів відеопам'яті у функції `GetScrSymbol()` використано довгий вказівник `addr`. Початково `addr` отримує адресу першого байта відеопам'яті – ця адреса формується за допомогою бібліотечного макроса `MK_FP()`, а потім встановлюється на задане знакомісце. Значенням виразу `*addr` є числове дане з типом `unsigned`, у молодшому байті якого записано ASCII-код символа, а в старшому – його атрибути. Байти коду та атрибутів символа можна легко виділити зі значення, яке повертає функція, використовуючи порозрядні операції. Наприклад:

```
   unsigned int  scr_symb, code, attr;
scr_symb = GetScrSymbol(symy, symx);
code = scr_symb & 0xff;                    /* код символа */
attr = scr_symb >> 8;                      /* атрибути символа */
```

Вищої швидкодії звертання до відеопам'яті можна досягти, застосовуючи спеціальні near-вказівники: `char _es*` та `unsigned _es*`. Перший з них використовують для побайтової роботи з відеопам'яттю, а другий – у разі запису/зчитування цілих слів. Нагадаємо, що ці вказівники зберігають тільки зміщення адреси символа, а сегментну частину адреси перед звертанням до відеопам'яті треба занести в системний регістр ES.

Записана далі функція `Invert()` візуально виділяє задану частину рядка екрана, інвертуючи двійкові коди атрибутів символів. Параметри `x1` та `x2` задають номери початкового та кінцевого символів ділянки інвертування у рядку `r`. Просування по байтах атрибутів у функції `Invert()` здійснюється через вказівник `patr`, що має тип `char _es*` і зберігає зміщення адрес.

```
/*  Функція інвертування атрибутів символів рядка  */
void  Invert (int r, int x1, int x2)
{
    char _es *patr=(char _es*)(((r-1)*80+x1-1)*2+1);
  _ES = 0xb800;                                  /* сегментна частина адрес */
  while (x1++ <= x2) {
    *patr = ~ *patr;                      /* інвертування байта атрибутів */
    patr += 2;
  }
}
```

Наступна функція `ScreenCopy()` виконує копіювання цілої відеосторінки (точніше, видимої на екрані частини). Сторінка, номер якої задається параметром `from_page`, переписується на іншу відеосторінку, що має номер `to_page`. Фактично функція копіює масив обсягом 25×80×2 байт, тому для неї питання швидкодії є особливо важливими. Оскільки сегментна частина адрес двобайтового слова, що копіюється, та ділянки, в яку воно записується, однакова (це адреса сегмента відеопам'яті), то найбільш ефективним способом адресації даних буде використання двох коротких `_es`-вказівників `pfrom` та `pto`.

```
/*  Функція копіювання відеосторінки  */
void  ScreenCopy (int from_page, int to_page)
 {
    unsigned _es *pfrom=(unsigned _es *)(from_page*4096);
    unsigned _es *pto=(unsigned _es *)(to_page*4096);
    int n;                              /* номер символа, що копіюється */
  _ES = 0xb800;                                          /* сегмент адрес */
  for (n=0;  n<25*80;  n++, pfrom++, pto++)
    *pto=*pfrom;                      /* копіювання символів з атрибутами */
 }
```

За допомогою функції `ScreenCopy()` можна швидко зберегти поточне екранне зображення на вільній відеосторінці, а потім відновити його на екрані, активізувавши дану відеосторінку або повторно скопіювавши її на нульову сторінку відеопам'яті.

14.5. Особливості звертання до динамічної пам'яті через функції Borland C

Як вже зазначалось, молодші та старші моделі розподілу пам'яті використовують різну динамічну пам'ять (див. табл. 14.1). Зокрема, модель Tiny може використовувати тільки внутрішню динамічну пам'ять, тобто незайняту кодом і даними область у межах єдиного програмного сегмента. Моделі Small і Medium теж мають внутрішню динамічну пам'ять, розташовану в межах сегмента даних. Крім цього вони можуть використовувати зовнішню динамічну пам'ять, яка займає вільну на даний момент частину оперативної пам'яті з адресного простору MS DOS (див. рис. 14.1). Старші моделі Compact, Large і Huge не мають внутрішньої динамічної пам'яті, вони можуть використовувати тільки зовнішню Heap-область (див. рис. 14.2).

Наявність двох видів динамічної пам'яті (внутрішньої та зовнішньої для малих моделей і тільки зовнішньої для старших моделей розподілу пам'яті) зумовлює особливості організації бібліотечних функцій Borland C, які реалізують звертання до динамічної пам'яті. Бібліотека Borland C містить також ряд додаткових функцій, які дають змогу визначати обсяг незайнятої динамічної пам'яті, здійснювати покроковий контроль пам'яті, перевіряти звільнені попередньо блоки пам'яті або змінювати обсяг блоку тощо. Прототипи цих функцій записані в заголовному файлі `<alloc.h>`.

У разі компіляції програми для молодших моделей розподілу пам'яті, стандартні бібліотечні функції `malloc()`, `calloc()`, `realloc()` та `free()`, описані в параграфі 13.1, використовують внутрішню динамічну пам'ять. Сумарний обсяг усіх ділянок, виділених у динамічній пам'яті цими функціями, не може перевищувати обсяг сегмента (64 Кбайт). Параметр обсягу пам'яті в цих функціях має тип `size_t` (тобто `unsigned int`). Всі функції працюють з короткими (`near`) вказівниками.

Якщо ж у програмі, що компілюється для моделі пам'яті Small або Medium, для збереження даних заплановано використовувати зовнішню динамічну пам'ять, то треба скористатись відповідними функціями з префіксом `far-`: `farmalloc()`, `farcalloc()`, `farrealloc()` та `farfree()`. Параметр обсягу пам'яті в far-функціях має тип `unsigned long`, що дає змогу виділяти динамічні ділянки, обсяг яких перевищує 64 Кбайт. Перелічені функції працюють з вказівниками довгого (`far`) формату. Наприклад, прототип функції `farmalloc()`, яка є аналогом функції `malloc()`, але призначена для роботи зі зовнішньою динамічною пам'яттю, оголошено так:

```
void far*  farmalloc (unsigned long memsize);
```

У програмах, розрахованих на молодші моделі розподілу пам'яті, вказівники, що використовуються для звертання до зовнішньої динамічної пам'яті, мають бути обов'язково оголошені з модифікаторами `far` або `huge`. Наприклад:

```
double far*  darr;
darr = (double far*)farcalloc(1000, sizeof(double));
```

Вказівник `darr` отримає адресу першого байта ділянки, розташованої у зовнішній динамічній пам'яті, в яку можна записати масив із 1000 дійсних чисел.

Якщо створюється динамічний масив, обсяг якого більший ніж 64 Кбайт (обсяг сегмента пам'яті), то для коректної адресації елементів такого масиву слід застосовувати `huge`-вказівники, щоб уникнути помилок адресної арифметики, властивих `far`-вказівникам у випадках міжсегментних переходів.

Для старших моделей розподілу пам'яті Compact, Large і Huge, які за замовчуванням працюють з довгими вказівниками і використовують тільки зовнішню динамічну пам'ять, можна застосовувати як стандартні функції без префікса `far-`, так і відповідні функції з цим префіксом. Обидві функції `malloc()` і `farmalloc()`, а також всі інші пари відповідних функцій у разі старших моделей виконують однакові дії та повертають однакове значення. Різниця між ними полягає тільки в обсязі ділянки, яка може бути виділена або звільнена функцією у зовнішній динамічній пам'яті. Для стандартних функцій обсяг не може перевищувати 64 Кбайт, а для функцій з префіксом `far-` обмеження накладає лише обсяг вільної динамічної пам'яті.

В `<alloc.h>` оголошено також ряд функцій, призначених для контролю незайнятої та звільненої частини пам'яті. Зокрема, обсяг вільної внутрішньої та зовнішньої динамічної пам'яті повертають функції `coreleft()`:

```
unsigned  coreleft(void);              /* для внутрішньої ДП  */
unsigned long  farcoreleft(void);      /* для зовнішньої ДП   */
unsigned long  coreleft(void);         /* для старших моделей */
```

Властивості функції `coreleft()`, а також функцій виділення і звільнення динамічної пам'яті у разі молодших і старших моделей пам'яті Borland C ілюструють результати реалізації поданої нижче демонстраційної програми. У програмі оголошено два вказівники `pmdl` та `pfar`, перший з яких отримує формат за замовчуванням відповідно до встановленої моделі пам'яті, а другий оголошено явно як `huge`-вказівник. Вказівник `pmdl`, який набуває адреси динамічної ділянки обсягом 10000 байтів, використовується для роботи з функціями без префікса `far-`. Відповідно в молодших моделях він буде пов'язаний з внутрішньою, а в старших – зі зовнішньою динамічною пам'яттю. Вказівник `pfar` призначений для роботи зі зовнішньою динамічною пам'яттю, він адресує ділянку, обсяг якої становитиме 150000 байтів. Допоміжний масив `arr` оголошено як глобальний, він займає 50000 байтів у сегменті даних програми.

```
/*************************************************************/
/*  Звертання до динамічної пам'яті у разі малих і великих   */
/*  моделей розподілу пам'яті системи програмування Borland C */
/*************************************************************/

#include <stdio.h>
#include <alloc.h>

char arr[50000];                    /* глобальний масив */

int main (void)
{
    char* pmdl;                     /* вказівник за замовчуванням */
    char huge* pfar;                /* дальній вказівник */
```

```
 printf("\n\tВільна динамічна пам\'ять (у байтах):\n"
        "  1) перед виділенням пам\'яті:\n");
 printf("внутрішня/зовнішня (за замовчуванням) - %6lu",
        (long)coreleft());
 printf("  зовнішня - %6lu", farcoreleft());
 pmdl =(char *)malloc(10000);                    /* виділення пам'яті */
 pfar =(char huge *)farmalloc(150000);
 printf("\n  2) після виділення пам\'яті:\n");
 printf("внутрішня/зовнішня (за замовчуванням) - %6lu",
        (long)coreleft());
 printf("  зовнішня - %6lu", farcoreleft());
 free(pmdl);     farfree(pfar);                  /* звільнення пам'яті */
 printf("\n  3) після звільнення пам\'яті:\n");
 printf("внутрішня/зовнішня (за замовчуванням) - %6lu",
        (long)coreleft());
 printf("  зовнішня - %6lu", farcoreleft());
 return 0;
}
```

Приклад виконання програми для моделі пам'яті Small:

```
    Вільна динамічна пам'ять (у байтах)
  1) перед виділенням пам'яті:
внутрішня/зовнішня (за замовчуванням) -  13488,  зовнішня - 532304
  2) після виділення пам'яті:
внутрішня/зовнішня (за замовчуванням) -   3488,  зовнішня - 382288
  3) після звільнення пам'яті:
внутрішня/зовнішня (за замовчуванням) -  13488,  зовнішня - 532304
```

Приклад виконання для моделі пам'яті Large:

```
    Вільна динамічна пам'ять (у байтах)
  1) перед виділенням пам'яті:
внутрішня/зовнішня (за замовчуванням) - 542032,  зовнішня - 542032
  2) після виділення пам'яті:
внутрішня/зовнішня (за замовчуванням) - 382000,  зовнішня - 382000
  3) після звільнення пам'яті:
внутрішня/зовнішня (за замовчуванням) - 542032,  зовнішня - 542032
```

Із наведених результатів видно, що для моделі пам'яті Small функції без префікса far- працюють з внутрішньою динамічною пам'яттю, яка займає вільний простір сегмента даних програми, де зберігається масив arr. Функції з префіксом far- програмують зовнішню динамічну пам'ять. Коли ж програму відкомпільовано для моделі пам'яті Large, то використовується тільки зовнішня динамічна пам'ять, в якій для потреб програми функції malloc() та farmalloc() виділяють дві ділянки, сумарний обсяг яких становить 10000+150000=160000 байтів.

Запитання та завдання для самоконтролю

1. Як формується фізична адреса оперативної пам'яті у межах початкових 1Мбайт? Які системні регістри використовуються для збереження сегментних компонентів адрес?
2. Що визначає модель розподілу пам'яті? Які моделі пам'яті підтримує середовище Borland C? Які з них називають молодшими, а які – старшими?
3. Який формат вказівників є базовим для молодших моделей пам'яті? Як у програмах, відкомпільованих для молодших моделей, можна звернутись до об'єктів, розташованих у інших сегментах?
4. З чого складаються довгі вказівники? Як сформувати `far`-вказівник зі сегментної адреси і зміщення? Які особливості `huge`-вказівників?
5. Які обмеження має застосування `far`-вказівників? Які вказівники треба використовувати для коректної роботи з масивом даних, обсяг якого перевищує обсяг сегмента?
6. Яке призначення спеціальних коротких вказівників, що оголошуються через модифікатори: `_cs`, `_ds`, `_es` та `_ds`?
7. Яку динамічну пам'ять: внутрішню і/або зовнішню може використовувати кожна з шести моделей розподілу пам'яті?
8. Як виділити місце для даних у зовнішній динамічній пам'яті, якщо програма буде компілюватися для моделі Small або Medium?
9. Яке значення повертають функції `coreleft()` та `farcoreleft()`, якщо вони виконуються у програмі, відкомпільованій для старших моделей пам'яті?
10. Напишіть коротку програму, яка перевіряє обсяг вільної динамічної пам'яті, після чого резервує (якщо це можливо) ділянку обсягом 30000 байтів і повторно визначає обсяг незайнятої пам'яті. Програма повинна використовувати функції роботи з динамічною пам'яттю без префікса `far`-. Встановіть модель розподілу пам'яті Small, відкомпілюйте і виконайте програму, зафіксуйте результати. Потім повторіть виконання програми для моделі Huge. Порівняйте результати. Яка динамічна пам'ять використовувалась у першій реалізації програми? Який обсяг цієї пам'яті? А в другій реалізації програми?
11. В яких випадках виведення текстової інформації доцільно здійснювати через безпосереднє звертання до відеопам'яті комп'ютера?
12. В якій формі зберігаються у відеопам'яті символи, що висвітлюються на екрані?
13. Як знайти адресу символа в оперативній пам'яті, якщо його знакомісце на екрані в текстовому режимі відображення задається координатами (x, y)?

Запрограмуйте наведені нижче задачі, використовуючи безпосереднє звертання до відеопам'яті текстових режимів

14. Виділити червоним кольором усі символи `*` в поточному екранному зображенні.
15. Скопіювати в буфер, попередньо виділений у динамічній пам'яті, зображення нижніх трьох рядків екрана, заповнивши їх символом `=`. Пізніше відновити збережені рядки.
16. Розробити функцію, що записує в заданий буфер слово з екрана, на якому в даний момент розміщений текстовий курсор. Параметри функції: координати екранного курсора та адреса буфера. Функція повинна повертати адресу записаного слова або `NULL`, якщо символ, під яким розміщений курсор, належить до символів-роздільників.

Розділ 15

ОБМІН ДАНИМИ З ФАЙЛАМИ

У цьому розділі:

- Три рівні організації введення/виведення даних
- Поняття файла і потоку; буферизація даних у процесах високорівневого потокоорієнтованого обміну даними
- Поділ бібліотечних функцій `<stdio.h>` на групи за призначенням
- Відкриття та закриття потоків, функції `fopen()` та `fclose()`
- Стандартні потоки, перескерування потоків
- Бібліотечні функції потокового введення/виведення окремих символів і символьних рядків
- Файловий обмін блоками даних, функції `fread()` та `fwrite()`
- Форматне введення/виведення даних, функції `fscanf()` та `fprintf()`
- Позиціювання покажчика поточної позиції файла
- Функції перевірки стану файла та аналізу помилок
- Функції роботи з буферами потоків
- Редагування вмісту файла, витирання та перейменування файлів, функції `<dir.h>` для роботи з каталогами та файлами
- Функції низькорівневого звертання до файлів

У компілятор мови С не включено спеціальних засобів для введення/виведення даних. Тому обмін даними як зі зовнішніми (консольними) пристроями, так і з дисковими файлами реалізовано через відповідні набори бібліотечних функцій. Це не тільки істотно розширює функціональні можливості системи введення/виведення даних, а й робить цю систему гнучкою та пристосованою до різних операційних середовищ і наявного апаратного забезпечення комп'ютера.

Бібліотеки більшості систем програмування мови С підтримують функції, що дають змогу здійснювати операції введення/виведення даних на трьох рівнях:

- високорівневе, т. зв. потокоорієнтоване введення/виведення;
- введення/виведення низького рівня;
- обмін даними з консольними пристроями.

Високорівневе введення/виведення використовує однаковий підхід у програмуванні обміну даними з файлами і зовнішніми пристроями та єдиний інтерфейс, незалежний від структури і способів доступу до файла чи термінального пристрою. Всі файли та дані з пристроїв розглядаються як неструктуровані набори байтів – *потоки*. Функції високорівневого потокового введення/виведення прості в програмуванні, мобільні, оскільки вони підтримуються стандартом мови С, та високоефективні завдяки внутрішній буферизації даних. Прототипи функцій потокоорієнтованого буферизованого обміну даними записані в заголовному файлі `<stdio.h>` (див. табл. Д2.5 у Додатку 2). Основні з них розглянемо в цьому розділі.

Функції низькорівневого введення/виведення базуються на засобах обміну даними, що властиві кожній конкретній операційній системі, тому вони не належать до стандартизованих. Ці функції не виконують форматування даних у процесах введення/виведення і не застосовують буферизації. З кожним файлом, відкритим на низькому рівні, пов'язується свій цілочисловий *дескриптор* (його також називають *префіксом*). Дескриптор зберігає номер позиції у внутрішніх таблицях операційної системи, де записано інформацію про цей файл. У всіх операціях обміну даними вказують дескриптор файла. Функції низькорівневого введення/виведення блокоорієнтовані і забезпечують виграш у швидкодії лише тоді, коли обсяг блоку даних, що передається за одну операцію введення чи виведення, кратний ємності сектора диска, що становить 512 байтів. Прототипи функцій бібліотеки Borland C, які реалізують низькорівневе звертання до файлів, зберігаються в заголовному файлі `<io.h>`.

Функції консольного введення/виведення доповнюють і розширюють можливості високорівневих функцій щодо введення даних з клавіатури і керування формою зображення екранних повідомлень у текстових режимах виведення інформації. До цієї групи належать також функції передавання даних через порти. Консольний обмін даними найбільш залежний від операційної та апаратної платформи комп'ютера. Прототипи функцій консольного введення/виведення оголошені в заголовному файлі `<conio.h>` (див. табл. Д2.7 у Додатку 2). Їх практичне застосування розглянемо в розділі 16.

15.1. Файли і потоки, буферизація даних

У мові С *файл* є достатньо абстрактним поняттям, яким позначається джерело надходження даних або місце їх нагромадження. Файлом вважається не тільки іменована сукупність даних, розташованих на зовнішньому носії (наприклад, дисковий файл), а й термінальний пристрій (наприклад, клавіатура або принтер). Це означає, що файли можуть мати зовсім різні характеристики. Наприклад, до даних, записаних у файлі на диску, можна виконувати прямий доступ, а введення даних з клавіатури може бути тільки послідовним.

Для уніфікації процесів файлового обміну даними у функціях високорівневого введення/виведення мови С використовують поняття *потоку* (*stream*), тобто послідовності байтів, що надходять від певного логічного пристрою (файла) або передаються у цей пристрій (файл). Щоб узагальнити обмін даними як для пристроїв, що підтри-

мують побайтове передавання (клавіатура, дисплей, принтер), так і для пристроїв, що передбачають обмін блоками байтів (дискові файли), у процесах введення/виведення здійснюється проміжна буферизація даних.

Буферизація полягає в наступному. Для кожного відкритого файла в оперативній пам'яті створюється буфер обміну заданої ємності (рис. 15.1). Здебільшого ємність буфера для дискових файлів приймається кратною ємності сектора диска (а саме: 512, 1024, 2048 байтів), щоб забезпечити максимальну ефективність операцій обміну. Якщо файл відкривається для читання, то буфер введення відразу заповнюється початковою порцією байтів цього файла. Коли всі дані з буфера зчитано, автоматично виконується наступне звертання до файла і в буфер заноситься нова порція байтів. Якщо ж здійснюється запис даних у файл, то вони спочатку заносяться у буфер виведення, а вже звідти переписуються у файл за кожної з умов: 1 – заповнено весь буфер, 2 – виконується очищення буфера, 3 – відбувається закриття файла, 4 – програма завершує роботу стандартним чином (неаварійно). Буферизація забезпечує високу швидкодію обміну, оскільки реально дані переносяться з однієї області оперативної пам'яті в іншу: з буфера в область даних програми або навпаки. Водночас буферизація мінімізує кількість звертань до фізичних пристроїв, які найбільше гальмують процеси введення/виведення даних.

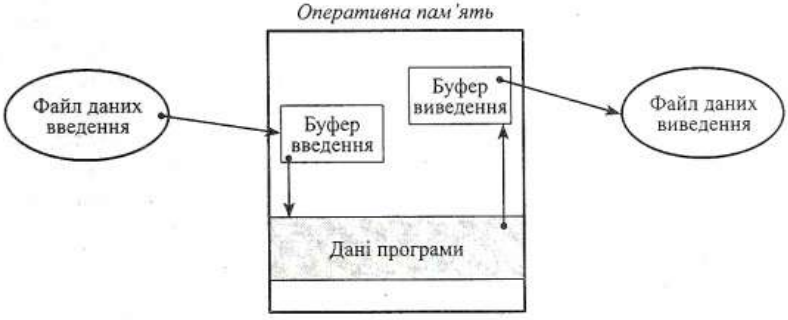


Рис. 15.1. Схема буферизації даних у процесах файлового введення/виведення

Підсумовуючи, відзначимо, що високорівневий потоковий обмін даними – це обмін байтами з фізичними файлами і логічними пристроями через систему буферизації, що дає змогу опрацьовувати дані різних форматів і розмірів.

15.2. Групи функцій для роботи з потоками

Всі бібліотечні функції високорівневого потокоорієнтованого буферизованого введення/виведення оголошені в заголовному файлі `<stdio.h>`. У цьому ж файлі визначені макроконстанти, внутрішні змінні, а також спеціальні структурні типи, що використовуються цими функціями.

Набір функцій для роботи з потоками можна поділити на групи:

- функції відкриття/закриття та перескерування потоків;
- функції введення/виведення даних;
- функції звертання до покажчика позиції файла;
- функції контролю стану файла та помилок введення/виведення;
- функції керування буферизацією;
- інші допоміжні функції.

Далі охарактеризуємо функції кожної з цих груп і наведемо приклади їх використання.

15.3. Відкриття/закриття потоків

Функція `fopen()`. Перед тим, як виконувати введення даних з конкретного файла або запис даних у файл, необхідно створити окремий потік, пов'язаний з цим фізичним файлом. Створення потоку реалізує функція відкриття файла, прототип якої оголошено так:

```
FILE * fopen (char * fname, char * fmode);
```

Функція створює новий потік і пов'язує його з фізичним файлом, заданим іменем `fname`. Параметр `fmode` задає режими обміну даними. За умови успішного відкриття потоку створюється спеціальна структура зі шаблоном `FILE`, оголошеним через декларацію `typedef` у заголовному файлі `<stdio.h>`. Поля цієї структури заповнюються інформацією про відкритий файл і створений потік, необхідною для організації файлового введення/виведення в конкретному операційному середовищі. Функція повертає адресу (вказівник) створеної та заповненої даними структури `FILE`. Якщо ж потік відкрити не вдалось (найчастіше це буває через неправильно задане ім'я файла або хибний шлях), то структура `FILE` не створюється, а `fopen()` повертає порожній вказівник `NULL`.

Зупинимось детальніше на параметрах функції `fopen()`. Обидва є вказівниками на символьні рядки. Перший параметр – `fname` задає ім'я файла. Пошук файла з даним іменем (якщо ім'я включає розширення, то його необхідно вказати) виконується тільки в тому каталозі, який є активним на даний момент. Для файлів з інших каталогів треба вказувати шлях за правилами відповідної операційної системи. Нагадаємо, що в константних символьних рядках мови С символ лівого слеша треба подвоювати, тобто шлях до файла

```
A:\LABROBC\delwords.cpp
```

у формі символьного рядка треба записувати так:

```
"A:\\LABROBC\\delwords.cpp"
```

Другий параметр функції `fopen()` – `fmode` задає спосіб відкриття файла (потоку) і характер наступних операцій обміну зі створеним потоком. Основними режимами відкриття є:

- `"r"` – файл відкривається для читання;
- `"w"` – файл відкривається для запису;
- `"a"` – файл відкривається для доповнення.

Режим `"r"` передбачає, що файл, який треба відкрити, вже існує, інакше фіксується помилка і `fopen()` повертає значення `NULL`. Режим `"w"` призначений для створення нового файла, якщо ж файл зі заданим іменем вже є в робочому каталозі, то його вміст буде витертий. Щоб унеможливити випадкове витирання однойменного файла, треба попередньо перевірити, чи такий файл існує (наприклад, спробувати відкрити його для читання), або скористатись однією з функцій генерації унікального імені файла (вони описані в параграфах 15.9 і 15.10). Режим `"a"` зберігає вміст існуючого файла, а якщо файл зі заданим іменем відсутній, то створює новий файл.

До ключової літери кожного режиму відкриття можна додати знак `+`, а саме: `"r+"`, `"w+"`, `"a+"`. У цьому випадку відкриті файли можна буде використовувати як для читання, так і для запису. Режим `"r+"` вимагає, щоб файл, який відкривається, вже існував, і не дозволяє збільшувати розмір файла в процесі запису. Файли, відкриті в режимах `"w+"` і `"a+"`, можна збільшувати, а інформацію, занесену в них, зчитувати.

Додатково в параметрі `fmode` можна вказати літеру `t` або `b`, якими задають відповідно текстовий чи бінарний (двійковий) режим відкриття потоку. За правилами замовчування автоматично встановлюється текстовий режим. Різниця між двома режимами полягає тільки в інтерпретації двобайтової комбінації символів з кодами `0xd` і `0xa` ("повернення каретки" і "новий рядок"), якою позначається натискання клавіші *Enter*. Якщо файловий обмін виконується в текстовому режимі, то дана комбінація в процесі читання замінюється одним символом нового рядка `'\n'`, а в разі запису навпаки – кожен символ `'\n'` заноситься у потік як двобайтова комбінація `"\r\n"`. У бінарних режимах такі заміни не виконуються. Оскільки у файлах, що містять двійкові коди числових даних, імовірна поява двох сусідніх байтів, значення яких відповідають комбінації `"\r\n"`, то, щоб уникнути їх перетворення, треба встановити бінарний режим відкриття потоку, наприклад: `"rb"` або `"w+b"` (останній режим можна позначити еквівалентною формою запису: `"wb+"`).

Наведемо приклад відкриття потоку з перевіркою коректності результату.

```
FILE *fp;                              /* вказівник потоку */
char *fname="report.txt";              /* ім'я файла */
fp=fopen(fname, "rt");
if (fp==NULL) {                        /* потік відкрити не вдалось */
   printf ("Файл %s не знайдено\n", fname);
   return 0;
}
. . .          /* дії з відкритим файлом */
```

Кожен потік повинен мати свій вказівник з типом `FILE*`, який зберігає адресу структури, заповненої даними про відкритий файл і пов'язаний з ним потік. Цей вказівник надалі ми називатимемо просто *вказівником потоку*.

Закриття потоків. У разі нормального (неаварійного) завершення роботи програми всі відкриті потоки автоматично закриваються, а їх буфери звільняються, проте найкраще програмно закривати ті потоки, які вже не використовуються. Для цього призначена функція

```
int  fclose (FILE* fp);
```

Функція має один параметр `fp` – вказівник на потік, який треба закрити. У разі успішного виконання `fclose()` повертає значення `0`. Якщо ж потік закрити не вдалось (це може трапитись через витирання файла або відсутність вільного місця на диску), то `fclose()` повертає макроконстанту `EOF`, оголошену в `<stdio.h>`.

У більшості систем програмування, в т.ч. у Borland C, значенням макроконстанти `EOF` є `–1`, але це значення не стандартизоване.

Система програмування Borland C додатково підтримує функцію

```
int  fcloseall (void);
```

яка закриває відразу всі потоки, відкриті в програмі на даний момент. Функція повертає кількість закритих потоків, а в разі помилки закриття файлів – `EOF`.

15.4. Стандартні потоки, перескерування потоків

Стандартні потоки С. На початку виконання кожної С-програми автоматично відкриваються три стандартні потоки:

- `stdin` – стандартний потік введення, який за правилами замовчування пов'язується з клавіатурою;
- `stdout` – стандартний потік виведення, який найчастіше пов'язується з виведенням даних на екран;
- `stderr` – стандартний потік повідомлень про помилки, який теж здебільшого скеровується на екран.

Ці потоки мають визначені імена, поза цим вони нічим не відрізняються від інших потоків, їх можна використовувати в усіх операціях файлового введення/виведення.

Перескерування потоку. Важливо, що стандартні потоки можна перескеровувати (перепризначати), тобто пов'язувати їх зі заданим файлом чи пристроєм. Перескерування підтримується не всіма операційними системами, але воно реалізоване в MS DOS, Windows, Unix, OS/2. Застосовуючи перескерування, можна простим способом записати в файл всі дані, які в програмі передаються у стандартний потік виведення, чи організувати зчитування з файла тих даних, які згідно з алгоритмом програми повинні надходити зі стандартного потоку введення.

Перескерування потоків виконує функція

```
FILE*  freopen (char* fname, char* fmode, FILE* fp);
```

Ця функція пов'язує потік `fp` з файлом, ім'я якого задає параметр `fname`, а режим доступу до даних – параметр `fmode`. Значення та форми задання параметрів `fname`

і `fmode` такі ж, як і у відповідних параметрів функції `fopen()`. За умови успішного виконання функція `freopen()` повертає вказівник на створений потік, а в разі помилки – константу `NULL`.

Слід пам'ятати, що після перескерування `stdout` всі операції виведення (зокрема, виведення підказок) будуть виконуватись не на екран, а в заданий файл. Аналогічно, в разі перескерування потоку введення `stdin`, всі дані, які за замовчуванням вводяться з клавіатури, будуть зчитуватись зі заданого файла. Стандартні потоки не треба ні відкривати, ні закривати – ці дії виконуються автоматично.

Приклад перескерування стандартного потоку виведення подано в наступній короткій програмі, де перше звертання до функції `puts()` викликає виведення повідомлення на екран, а друге – у файл example.out.

```
/*********************************************************/
/*  Приклад перескерування стандартного потоку виведення  */
/*********************************************************/

#include <stdio.h>

int  main (void)
{
   puts("\n\t Це повідомлення виводиться на екран.\n");
   freopen("example.out", "w", stdout);    /* потік  скеровано в файл */
   puts("Дане повідомлення буде записане в файл.\n");
   return 0;
}
```

MS DOS дозволяє також здійснювати перескерування стандартних потоків з командного рядка. Для цього використовують символи < та >, за якими вказують ім'я файла перескерування. Наприклад, запуск виконавчого коду програми countmax.exe з командного рядка у формі:

```
countmax.exe <numbers.dat >results.dat
```

вказує, що в процесі роботи програми countmax.exe дані, які мають вводитись зі стандартного вхідного потоку, будуть зчитуватись з файла numbers.dat, а дані, які виводяться у стандартний потік виведення, будуть записуватись у файл results.dat.

15.5. Функції потокового введення/виведення даних

Функції високорівневого обміну даними з файлами можна поділити на чотири групи відповідно до формату даних, що передаються за одну операцію читання/запису:

- функції посимвольного введення/виведення;
- функції рядкового введення/виведення;
- функції блокового введення/виведення;
- функції форматного введення/виведення.

Всі операції введення/виведення даних виконуються, починаючи з поточної позиції файла, яку зберігає спеціальний покажчик. У процесі введення цей покажчик автоматично зсувається на відповідну кількість байтів у напрямку до кінця потоку і встановлюється на перший ще непрочитаний байт. У процесі виведення даних покажчик поточної позиції файла автоматично встановлюється за останнім записаним байтом.

15.5.1. Посимвольний обмін даними

Введення одного символа. Основною функцією зчитування одного символа з потоку введення `fp` є функція

```
int  fgetc (FILE* fp);
```

У разі успішного виконання функція `fgetc()` повертає код зчитаного символа як значення з типом `unsigned char`, розширене до формату `int`. Якщо ж читання з потоку недоступне, зокрема через досягнення кінця файла, то функція повертає макроконстанту `EOF`.

Такі ж дії виконує функція

```
int  getc (FILE* fp);
```

яку в середовищі Borland C та в ряді інших систем програмування оголошено як макрос, що викликає функцію `fgetc()`.

Якщо параметром названих функцій є стандартний потік введення `stdin`, то їх дія повністю збігається з дією функцій введення символів з клавіатури `fgetchar()` і `getchar()`. Результат виклику кожної з чотирьох наступних функцій однаковий:

```
fgetc(stdin) ≡ getc(stdin) ≡ fgetchar() ≡ getchar()
```

– зчитування одного символа, введеного з клавіатури і записаного в буфер стандартного потоку введення (якщо потік `stdin` попередньо не перескерували).

Функція

```
int  ungetc (int symb, FILE* fp);
```

дає змогу повернути в потік введення `fp` останній зчитаний символ. Після повернення `symb` стає першим символом потоку, його буде зчитано довільною наступною операцією введення. Функція `ungetc()` повертає код символа `symb` у разі успішного завершення, а разі невдачі – значення `EOF`.

Наведемо приклад функції, яка реалізує зчитування послідовності цифрових символів зі заданого відкритого потоку `frd` та запис введених цифр за адресою `numbst`. Робота функції завершується, коли зчитується символ, який не є цифрою – цей символ функція повертає назад у потік.

```
/*  Функція введення послідовності цифрових символів  */
#include <stdio.h>
#include <ctype.h>
char*  ReadNumber (char* numbst, FILE* frd)
{
```

```
    char *pn=numbst;
    int symb;
  while (isdigit(symb=getc(frd)))
     *pn++=symb;                    /* запис цифрових символів у numbst */
  *pn='\0';                                  /* кінець числового рядка */
  ungetc(symb, frd);                 /* повернення нечислового символа */
  return numbst;
}
```

Запис символа у файл. Запис заданого символа `symb` у потік виведення `fp` виконують функції

```
int  fputc (int symb, FILE* fp);
```

та

```
int  putc (int symb, FILE* fp);
```

Другу з цих функцій у Borland C реалізовано як макрос. Обидві функції за умови успішного виконання повертають код записаного символа. У разі помилки звертання до файла функції повертають значення макроконстанти `EOF`.

Проілюструємо використання функцій посимвольного потокового введення/виведення прикладом програми, яка відображає на екрані весь вміст текстового файла.

```
/************************************************/
/*  Посимвольне виведення вмісту заданого файла  */
/************************************************/
#include <stdio.h>
#include <string.h>
int  main (int argc, char** argv)
{
    FILE  *fp;
    char  fname[50];
    int  c;
  if (argc==1) {            /* ім'я файла у командному рядку не задано */
     printf("Ім\'я файла: ");
     gets(fname);
  }
  else
     strcpy(fname, argv[1]);     /* ім'я зчитується з командного рядка */
  fp=fopen(fname,"r");
  if (fp==NULL) {
     printf("Хибне ім\'я файла - %s \n", fname);
     return 0;
  }
  while ((c=getc(fp))!=EOF)                  /* цикл зчитування файла */
     putc(c, stdout);            /* посимвольне відображення на екрані */
  fclose(fp);
  return 0;
}
```

Наведена вище програма дає змогу задати ім'я файла з командного рядка або ввести його в процесі виконання програми. Хоча зчитування файла здійснюється посимвольно, все ж програма забезпечує високу швидкодію обміну, оскільки реально символи зчитуються не з файла на зовнішньому носії, а з буфера введення в оперативній пам'яті. Недолік програми в тому, що для великих за розміром файлів на екрані збережеться тільки остання сторінка тексту, а всі решта будуть витерті автоматичним скролінгом. Приклад посторінкового виведення файла подано далі.

Відображення символів, зчитаних з файла, у програмі реалізовано як посимвольне виведення у стандартний потік `stdout`. Загалом, виведення символа `symb` на екран (або в файл, якщо `stdout` попередньо перескерували) можна виконати кожним із чотирьох наступних викликів функцій:

```
fputc(symb, stdout);        putc(symb, stdout);
fputchar(symb);             putchar(symb);
```

15.5.2. Файловий обмін рядками символів

Зчитування рядка символів з потоку введення виконує функція

```
char* fgets (char* str, int max, FILE* fp);
```

На відміну від `gets()`, дана функція має три параметри: перший – `str` вказує на масив символів, в який буде записано введений рядок; другий – `max` задає максимальну кількість символів (з нуль-символом включно), яку може містити зчитаний рядок; останній параметр – `fp` вказує на потік, з якого вводяться рядки.

У разі успішного виконання функція `fgets()` повертає адресу першого символа введеного рядка, тобто `str`, а в разі невдачі, зокрема, якщо досягнуто кінця файла, – вказівник `NULL`.

Звернемо увагу на кілька важливих особливостей цієї функції. Параметр `str` має вказувати на ділянку оперативної пам'яті, обсяг якої достатній для збереження зчитаного рядка. Параметр `max` обмежує кількість символів, які вводяться з потоку, тому розмір масиву `str` має бути не меншим за `max`. Дія функції така: з потоку `fp` у ділянку оперативної пам'яті, на яку вказує `str`, зчитується послідовність символів до символа нового рядка чи символа кінця файла, але не більше, ніж `max-1` символів. Якщо в процесі введення зчитується символ `'\n'`, то він заноситься у `str`, за ним записується `'\0'` і процес введення припиняється. У противному разі в `str` послідовно заноситься `max-1` символів потоку, за якими записується нуль-символ. Треба пам'ятати, що в разі зчитування з файла цілого рядка або його кінцевої частини, в `str` перед `'\0'` буде записано символ `'\n'`, якщо потік відкрито в текстовому режимі, або пару символів `"\r\n"`, якщо потік відкрито як бінарний.

Запис заданого рядка в потік виведення здійснює функція

```
int fputs (char* str, FILE* fp);
```

яка повертає ненульове значення за умови успішного виконання та `EOF` у разі невдачі.

У процесі виконання `fputs()` послідовно передає у потік `fp` символи рядка `str` до `'\0'`. Сам нуль-символ у потік не записується і не перетворюється в інші символи. Якщо в рядку зустрічається символ нового рядка, а потік виведення текстовий, то замість символа `'\n'` у потік записується комбінація символів `"\r\n"`. Для бінарних потоків жодні заміни символів не виконуються.

Наведемо приклад використання функцій введення/виведення символьних рядків для посторінкового роздруку заданого файла. У програмі виконується підрахунок кількості рядків, виведених на екран. Після відтворення сторінки тексту встановленого розміру, читання з файла затримується до натискання на довільну клавішу.

```
/*******************************************/
/*  Посторінкове виведення текстового файла  */
/*******************************************/

#include <stdio.h>
#include <conio.h>                    /* для функцій clrscr() і getch() */
#define  LEN  81                     /* розмірність буфера рядків тексту */
#define  NROWS  22                        /* розмір сторінки виведення */
int  main ()
{
    FILE  *f;
    char  buf[LEN];
    char  fname[] = "newtext.txt";
    int  n = 0;
  if ((f = fopen(fname, "rt")) == NULL) {
      printf("\n\t Файл %s не знайдено\n", fname);
      return 0;
  }
  clrscr();                                          /* очищення екрана */
  n = 0;
  while (fgets(buf, LEN, f) != NULL) {
     fputs(buf, stdout);                  /* відображення зчитаного рядка */
     if (++n % NROWS == 0) {                          /* заповнено сторінку */
         printf("\n\t\tСторінка %d.\n", n/NROWS);
         getch();                                    /* затримка зображення */
     }
  }
  printf("\n\t\tКінець файла.\n");
  getch();
  fclose(f);
  return 0;
}
```

У цій програмі розмірність рядка введення `LEN` дорівнює розміру екранного рядка – 80 символів (плюс байт на `'\0'`), що полегшує підрахунок кількості висвітлених на екрані рядків.

Для виведення символьних рядків попередня програма використовує функцію

```
fputs(buf, stdout);
```

щоб відобразити на екрані текст так, як він записаний у файлі.

Якби виведення зчитаних з файла рядків виконувалось функцією `puts(buf)`, то після кожного рядка, що завершується символом `'\n'`, на екран додатково виводився б порожній рядок, оскільки `puts()` замінює кінцевий нуль-символ стрінга символом `'\n'`.

15.5.3. Обмін блоками даних

Функції `fread()` та `fwrite()`. Зчитати з потоку блок даних заданого розміру можна за допомогою функції

```
size_t  fread (void* buf, size_t size, size_t n, FILE* fp);
```

що заносить у буфер, адресу якого задає параметр `buf`, `n` об'єктів, кожен з яких має розмір `size`. Тип `size_t` оголошено в `<stdio.h>` через декларацію `typedef`. У Borland C він збігається з типом `unsigned int`.

Функція `fread()` повертає кількість реально зчитаних об'єктів (не байтів). Якщо ця величина не дорівнює заданому значенню `n`, то це означає, що досягнуто кінця файла або зафіксовано помилку читання. Уточнити причину неповного зчитування блоку можна за допомогою функцій `feof()` та `ferror()`, які опишемо далі.

Використання функції `fread()` проілюструємо прикладом програми, яка визначає середньоарифметичне значення набору дійсних чисел, бінарні коди яких зберігаються у файлі attempj.dat, зареєстрованому в каталозі C:\EXPRDATA. Дані з файла зчитуються в буфер блоками по `N` чисел. Процес роботи з файлом завершується, коли розмір зчитаного блоку менший за `N`, що сигналізує про досягнення кінця файла.

```
/*****************************************************************/
/*  Визначення середньоарифметичного значення послідовності     */
/*  дійсних чисел, що зберігається у заданому бінарному файлі   */
/*****************************************************************/
#include <stdio.h>
#define  N  1000                                  /* розмір блоку даних */
double  buf[N];
int  main (void)
{
    FILE  *fd;
    char  fname[] = "C:\\EXPRDATA\\attempj.dat";
    double  sum;
    unsigned  n, k, i;
  if ((fd=fopen(fname,"rb"))== NULL) {       /* бінарний режим обміну */
     printf("\n\t Файл %s не знайдено\n", fname);
     return 0;
  }
```

```
  sum=0;   k=0;            /* сума і кількість введених з файла чисел */
  do {
     n=fread(buf, sizeof(double), N, fd);
     k+=n;
     for (i=0;  i<n;  i++)
        sum+=buf[i];              /* сумування чисел введеного блоку */
  } while (n==N);
  printf("\nСереднє значення %u елементів файла %s => %.3lf\n",
            k, fname, sum/k);
  fclose(f);
  return 0;
}
```

Запис блоку даних у потік виконує функція

```
size_t  fwrite (void* buf, size_t size, size_t n, FILE* fp);
```

Параметри цієї функції такі ж, як і в функції `fread()`: `buf` – вказівник на ділянку оперативної пам'яті, звідки зчитуються дані; `size` – розмір одного даного; `n` – кількість даних, що мають бути записані у файл (потік); `fp` – вказівник потоку виведення.

Функції `getw()` та `putw()`. Бібліотека системи програмування Borland C додатково включає дві функції: `getw()` і `putw()`. Перша з них призначена для зчитування з бінарного файла двобайтового двійкового коду цілого числа. Друга функція записує в файл, відкритий у бінарному режимі, внутрішній двійковий код заданого цілочислового значення. Ці функції оголошені в `<stdio.h>` наступним чином:

```
int  getw (FILE* fp);
int  putw (int numb, FILE* fp);
```

У разі успішного виконання обидві функції повертають значення зчитаного або відповідно записаного числа, а в разі виникнення помилки – макроконстанту `EOF`.

Подана нижче функція `FindNumber()` здійснює пошук заданого числа `number` у відкритому файлі, заповненому бінарними кодами цілих чисел. Пошук розпочинається з поточної позиції файла. Функція повертає порядковий номер `number` або 0, якщо такого числа у файлі немає (припускаємо, що `number` не дорівнюватиме `EOF`).

```
/*  Функція пошуку заданого цілого числа в бінарному файлі  */
long  FindNumber (FILE* fb, int number)
{
    int rnum;   long k = 0;
  do {
     rnum = getw(fb);   k++;                  /* зчитування числа */
     if (rnum == number)
        return k;                             /* число знайдено */
  } while (rnum != EOF);
  return 0;                                   /* задане число відсутнє */
}
```

15.5.4. Форматне введення/виведення даних

Функція `fscanf()`. Файлове введення даних згідно зі заданим списком форматних специфікацій здійснює функція

```
int  fscanf (FILE* fp, char* format, ...);
```

Перший параметр функції – `fp` вказує на текстовий потік введення, другий обов'язковий параметр – `format` задає символьний рядок, в якому записано послідовність специфікацій форматних перетворень. Наступні параметри задають адреси змінних і/або ділянок пам'яті, куди будуть записуватись введені значення. Ці параметри не обов'язкові, їх кількість і типи визначаються специфікаціями рядка `format`.

Правила форматних перетворень даних, введених з потоку `fp`, та їх взаємоузгодження зі списком адрес введення такі ж, як і для функції `scanf()` (див. параграф 5.2). Тому, якщо вказівником потоку в `fscanf()` є стандартний потік `stdin`, то така функція цілком відповідає функції форматного введення `scanf()`.

Функція `fscanf()` повертає кількість успішно введених даних. Перевірка значення, яке повертає функція, дає змогу встановити, як пройшов процес введення: успішно, а чи виявлено помилку. Причину помилки можна конкретизувати через функції `feof()` та `ferror()`. Наведемо фрагмент програми, в якому здійснюється зчитування даних із текстового файла, заповненого цілими числами.

У першій версії програми некоректно використано функцію `feof()` для організації циклу введення даних, що може призвести до помилкових результатів.

```
/*  Приклад неправильної організації форматного введення даних  */
    FILE *fin;
    int numb;
  . . .                                  /* операції відкриття файла */
  while (!feof(fin)) {                   /* поки не досягнуто кінця файла */
    fscanf(fin, "%d", &numb);
    printf("%3d", numb);
  }
```

Якщо вміст файла введення є таким:

```
1_2_3_4_5
```

де символом підкреслення позначено один або декілька символів-роздільників (пробіл, горизонтальна табуляція чи новий рядок), то на екран у процесі виконання оператора циклу буде виведено:

```
1  2  3  4  5
```

Якщо ж у файлі після останнього числа буде записано ще хоча б один символ-роздільник:

```
1_2_3_4_5_
```

то результат читання та виведення на екран буде іншим:

```
1 2 3 4 5 5
```

Повторення останнього числа, зчитаного з файла, зумовлене тим, що після введення числа 5 ознака кінця файла не встановлюється через наявність у потоці кінцевих незчитаних розділових символів. Наступний виклик `fscanf()` зчитує ці символи, але не знаходить більше чисел, тому фіксується кінець файла, а значення `numb` не змінюється. Якби в кінці файла були записані якісь символи, що не є цифрами і роздільниками, наприклад:

```
1_2_3_4_5_КІНЕЦЬ
```

то це призвело б до зациклення наведеної програми, оскільки функція `fscanf()` не зчитує символів літер і повертає їх у потік, отже ознаку кінця файла не буде встановлено взагалі.

Щоб зробити форматне введення незалежним від прикінцевих символів файла, цикл читання даних слід записати, наприклад, так:

```
/*  Приклад правильної організації введення даних  */
  . . .                                  /* початкова частина програми */
  while (fscanf(fin, "%d", &numb)== 1)                /* поки є числа */
     printf("%3d", numb);
  if (feof(fin))
     puts("Зчитано всі числа файла");
  else
     puts("Помилка в числових даних");
```

Для читання символьних даних у функціях `fscanf()` і `scanf()` використовують специфікатор `"%s"`. Нагадаємо, що при цьому зчитується послідовність символів до першого роздільника. В разі звертання до текстового файла:

```
fscanf(f, "%s", str);
```

у рядок `str` буде зчитане тільки одне поточне слово зі заданого файла. Необхідно забезпечити, щоб обсяг ділянки, на яку вказує `str`, був достатнім для запису найдовшого слова файла.

Функція `fprinf()`. Форматне виведення даних здійснює функція

```
int  fprintf (FILE* fp, char* format, ...);
```

Функція має два обов'язкові параметри: перший – вказівник потоку виведення `fp`, а другий – текстовий рядок `format`, що містить специфікації форматних перетворень. Далі записуються вирази, значення яких мають бути виведені в потік у формі, що задається відповідною специфікацією рядка формату. Правила формування рядка `format` і взаємоузгодження його зі списком виведення такі ж, як і в функції `printf()` (див. параграф 5.1).Так само `fprintf()` повертає кількість символів, виведених у потік `fp`.

Наведена далі програма використовує функцію `fprintf()` для запису в файл з іменем fun_tab.res результатів табулювання заданої функції однієї змінної.

```
/*****************************************************************/
/*  Створення файла з результатами табулювання заданої функції  */
/*****************************************************************/
#include <stdio.h>
double  fun (double);                          /* функція, що табулюється */
int  main(void)
{
    FILE *fout;
    double x0, xk, x, dx;
  printf("\n Межі та крок табулювання:  ");
  scanf("%lf%lf%lf", &x0, &xk, &dx);
  fout=fopen("fun_tab.res", "wt");      /* створення текстового файла */
  for (x=x0;  x<xk;  x+=dx)                  /* цикл заповнення файла */
     fprintf(fout, "%10.2lf%15.3lf\n", x, fun(x) );
  fclose(fout);
  printf("\n Файл результатів табулювання створено. \n");
  return 0;
}
double  fun (double x)
{
   . . .                                  /* тіло функції, яка табулюється */
}
```

15.6. Встановлення поточної позиції файла

Високорівневий потокоорієнтований обмін даними з файлами організовано як передавання необхідної послідовності байтів. Кожна операція введення даних з файла, відкритого для читання, пов'язана зі зміщенням внутрішнього покажчика (індикатора) поточної позиції файла на відповідну кількість байтів. Файл, відкритий для запису (режим відкриття `"w"`), спочатку порожній, а в процесі запису покажчик поточної позиції автоматично встановлюється за останнім записаним байтом. Якщо ж файл відкривається для доповнення (режим відкриття `"a"`), то вміст файла зберігається, а покажчик поточної позиції встановлюється перед маркером кінця файла.

Хоча в процесах обміну даними послідовне зчитування з файла чи послідовний запис у файл застосовуються найчастіше, все ж у багатьох задачах потрібно керувати значенням покажчика поточної позиції файла. Це, зокрема, дає змогу виконувати операції введення/виведення даних, починаючи зі заданого байта файла.

Основною функцією позиціювання потоку даних є функція

```
int  fseek (FILE* fp, long offset, int base);
```

Функція `fseek()` встановлює покажчик поточної позиції файла, пов'язаного з потоком `fp`, відповідно до значень, заданих параметрами `offset` і `base`. Параметр `offset`, який має тип довгого цілого, задає кількість байтів, на які треба пересунути поточну позицію файла. Якщо `offset>0`, то покажчик поточної позиції зсувається у

напрямку кінця файла, коли ж `offset < 0`, то покажчик зсувається до початку файла. Параметр `base` задає базис, відносно якого здійснюється переміщення поточної позиції. Він може набувати значення однієї з трьох макроконстант:

- `SEEK_SET` (у Borland C ця макроконстанта має значення `0`) – за базис береться початок файла;
- `SEEK_CUR` (або `1`) – за базис береться поточна позиція файла;
- `SEEK_END` (або `2`) – за базис береться кінець файла.

У разі успішного виконання функція `fseek()` повертає нуль, а в разі виникнення помилки позиціювання – ненульове значення.

Наступне звертання встановлює покажчик поточної позиції на початок файла, пов'язаного з потоком `frw`:

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

Щоб встановити покажчик поточної позиції бінарного файла, заповненого двійковими кодами дійсних чисел, перед десятим елементом, рахуючи від кінця файла, треба викликати функцію `fseek()` з такими параметрами:

```
fseek(fdn, -10*sizeof(double), SEEK_END);
```

Ще одним способом швидкого переміщення покажчика поточної позиції відкритого файла на початок цього файла є використання функції

```
void  rewind (FILE* fp);
```

Значення покажчика поточної позиції файла можна отримати через функцію

```
long  ftell (FILE* fp);
```

Якщо потік `fp` відкрито в бінарному режимі, то значення, яке повертає `ftell()`, дорівнює кількості байтів від початку файла до його поточної позиції, зафіксованої у покажчику. Для текстових потоків це значення може бути неточним через перетворення символів кінця рядка. У разі помилки виконання функція `ftell()` повертає `-1L`.

Зберегти значення поточної позиції файла можна також за допомогою функції

```
int  fgetpos (FILE* fp, long* fpos);
```

Функція записує в змінну, адресу якої задає вказівник `fpos`, значення покажчика поточної позиції файла, пов'язаного з потоком `fp`. Повертає нуль у разі успішного завершення і ненульове значення – у разі виникнення помилки.

Для відновлення значення поточної позиції файла, збереженого раніше через звертання до `fgetpos()`, призначена функція

```
int  fsetpos (FILE* fp, long* fpos);
```

параметри і значення якої такі самі, як і в попередньої функції.

Якщо потік відкрито для запису й читання одночасно (режим відкриття включає символ `'+'`), то, щоб перейти від операцій запису до операцій читання або навпаки, треба обов'язково між цими операціями викликати одну з функцій позиціювання: `rewind()`, `fseek()` або `fsetpos()`.

Проілюструємо використання описаних функцій позиціювання на прикладі функції `FileSize()`, яка визначає розмір (у байтах) файла, відкритого в бінарному режимі.

```
/*  Визначення розміру заданого файла  */
long  FileSize(FILE * fz)
{
    long fsize, fpos;
  fgetpos(fz, &fpos);            /* збереження поточної позиції файла */
  fseek(fz, 0L, SEEK_END);                  /* перехід у кінець файла */
  fsize=ftell(fz);                       /* розмір відповідає зміщенню */
  fsetpos(fz, &fpos);  /* повернення вказівника на попередню позицію */
  return fsize;
}
```

15.7. Функції аналізу помилок

Більшість функцій файлового обміну даними повертають значення, яке дає змогу перевірити, чи дана операція пройшла успішно, чи відбувся збій. Встановити конкретну причину збою можна через функції аналізу помилок файлового введення/виведення.

Зокрема, функція

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

перевіряє, чи досягнуто кінця файла, пов'язаного з потоком `fp`. Повертає нуль, якщо не встановлено ознаку кінця файла, інакше – ненульове значення. Всі операції читання з файла після досягнення його кінця вважаються помилковими. Ознаку (індикатор) кінця файла знімають функції позиціювання `fseek()`, `rewind()`, `fsetpos()` та функція скидання індикаторів помилок `clearerr()`.

Організовуючи в С-програмах цикли введення даних з файла, треба пам'ятати, що ознака досягнення кінця встановлюється тільки тоді, коли зчитується код кінця файла. Раніше вже наводився приклад некоректного використання значення `feof()` як умови завершення зчитування з файла числових даних за допомогою функції `fscanf()`. Неправильною буде також наступна організація циклу введення двійкових кодів послідовності структур, які мають шаблон `struct data` і зчитуються у буфер `buf` з потоку `fdata`:

```
do {
     fread(&buf, sizeof(struct data), 1, fdata);
     . . .                          /* опрацювання введеної структури */
} while (!feof(fdata));
```

Помилка полягає в тому, що остання структура з файла даних буде опрацьовуватись двічі. Після її введення цикл не завершується, оскільки ще не зчитано код кінця файла, а тому `feof()` повертає `0`. Наступна операція читання не вводить нових даних, а тільки встановлює ознаку кінця файла, тому вміст `buf` не змінюється, а попередньо введена структура опрацьовується другий раз.

Функція

```
int ferorr (FILE* fp);
```

перевіряє, чи встановлено ознаку помилки в попередніх операціях звертання до потоків даних. Повертає нуль, якщо помилку не зафіксовано, та ненульове значення, якщо виявлено помилку.

Причину збою в операціях потокового введення/виведення можна розшифрувати, використовуючи функцію

```
void perorr (char* errtext);
```

Функція виводить у стандартний потік помилок `stderr` (звичайно це екран) текст повідомлення, записаний в `errtext`, а за ним після двокрапки стандартне системне пояснення причини виникнення помилки.

Наведемо приклад аналізу помилок під час роботи з потоком `fp`:

```
if (ferorr(fp))
    perorr( "Збій у роботі з файлом даних" );
```

Іншим способом розшифрування помилки є звертання до внутрішньої змінної `errno`, оголошеної в заголовному файлі `<errno.h>`. У процесі виконання програми в цю змінну автоматично заноситься номер кожної зафіксованої помилки потокового обміну. Щоб отримати текстове пояснення помилки з номером `errno`, треба скористатись функцією `strerror()`, оголошеною в `<string.h>`. У наведеному далі прикладі виконуються такі ж дії, як і в попередньому звертанні до функції `perorr()`, але додатково виводиться ім'я файла, для якого зафіксовано помилку:

```
if (errno)                          /* errno≠0 - зафіксовано помилку */
    fprintf(stderr, "Збій у роботі з файлом %s - %s \n",
            filename, strerror(errno));
```

Встановлені індикатори помилок і кінця файла скидає в нуль функція

```
void clearerr (FILE* fp);
```

У наступному фрагменті програми виклик функції `clearerr()` знімає ознаку кінця файла, щоб доповнити його вміст рядком поточної дати і часу.

```
FILE *f;
time_t curt;                        /* змінна для роботи з функціями часу */
   . . .          /* відкриття файла у режимі "r+t" і опрацювання його */
if feof(f) {                                    /* опрацьовано весь файл */
   clearerr(f);                              /* скидання індикатора помилок */
   time(&curt);                           /* визначення поточної дати і часу */
    /* занесення дати і часу в формі символьного рядка у кінець файла */
   fprintf(f, "\nФайл опрацьовано: %s", ctime(&curt));
}
```

Для визначення та виведення поточної дати й часу використано функції `time()` та `ctime()`, оголошені в заголовному файлі `<time.h>` (див. табл. Д2.6 у Додатку 2).

Перша з них повертає значення, що дорівнює кількості секунд від 1 січня 1970 року, а друга перетворює це значення у стандартний символьний рядок, що включає скорочені найменування дня тижня і місяця, номер дня, час у формі години:хвилини:секунди та поточний рік – DDD MMM dd hh:mm:ss YYYY.

15.8. Керування буферизацією даних

Нагадаємо, що високорівневий потокоорієнтований файловий обмін даними використовує проміжну буферизацію: дані з файла заносяться спочатку у внутрішній буфер, а вже звідти зчитуються програмою (див. рис. 15.1). Якщо ж здійснюється запис у файл, то спочатку дані заносяться в буфер виведення, звідки переписуються у файл, коли буфер заповнено, або файл закривається. Проміжна буферизація може спричинити втрату даних у випадку некоректного завершення роботи програми: зациклення, збій у системі, відключення живлення тощо. Тому важливу інформацію доцільно відразу переписувати з буфера у файл, не чекаючи автоматичного скидання буфера.

Очищення/переписування внутрішнього буфера потоку `fp` виконує функція

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

Якщо потік `fp` відкрито для запису, то `fflush()` переносить у файл, пов'язаний з `fp`, увесь вміст відповідного буфера запису. Якщо ж `fp` є потоком введення, то `fflush()` очищає буфер цього потоку. В обох випадках потоки залишаються відкритими.

У разі успішного завершення функція `fflush()` повертає нуль, а в разі помилки – значення макроконстанти `EOF`.

Подамо приклад скидання даних у файл одразу після операції запису.

```
/*  Перенесення у файл даних з буфера виведення  */
   FILE* fout;
   . . .                              /* відкриття потоку fout для запису */
do {
       . . .                                 /* формування даних виведення */
    fwrite(&inform, sizeof(inform), 1, fout);        /* запис у буфер */
    fflush(fout);                  /* перенесення даних з буфера в файл */
} while (...);                                    /* умова виконання циклу */
```

Для потоків введення, серед яких `stdin`, функцію `fflush()` застосовують, щоб звільнити буфер введення від зайвих даних чи символів. Таке очищення необхідне, зокрема, коли після введення даних функцією `fscanf()` чи `scanf()` наступною є операція читання символьного рядка за допомогою функцій `fgets()` або `gets()`.

Наведемо приклад очищення буфера у процесі введення даних з клавіатури.

```
/*  Звільнення буфера введення від зайвих символів  */
struct data {
   long index;
   char addr [120];
} inf;
```

```
. . .                                   /* початкова частина програми */
printf ("Індекс - ");      scanf("%ld", &inf.index);
fflush(stdin);                          /* очищення буфера перед gets() */
printf ("Адреса - ");      gets(inf.addr);
```

Якби в наведеному програмному фрагменті оператора `fflush(stdin);` не було, то функція `gets()` зчитала би порожній рядок, оскільки після попереднього введення індекса за допомогою функції `scanf()` у вхідному буфері залишаються всі кінцеві нецифрові символи (щонайменше – `'\n'`).

Програміст може встановити власний буфер потоку або відмінити буферизацію, використовуючи функцію

```
void  setbuf (FILE* fp, char* fbuf);
```

Якщо значенням параметра `fbuf` є `NULL`, то буферизація даних для вказаного потоку виконуватись не буде. Інакше `fbuf` повинен вказувати на ділянку пам'яті (або масив) обсягом `BUFSIZ`. Макроконстанту `BUFSIZ` оголошено в `<stdio.h>`, її значення у Borland C дорівнює 512.

Ширші можливості щодо керування буферизацією в процесах файлового обміну даними має функція

```
int  setvbuf (FILE* fp, char* fbuf, int bmode, size_t bsize);
```

яка дає змогу не тільки встановити для потоку власний буфер, а й вказати його ємність і режим роботи. Параметр `fbuf` задає адресу ділянки оперативної пам'яті, що буде використовуватись як буфер потоку `fp`. Обсяг цієї ділянки повинен бути не меншим за `bsize`. Якщо `fbuf` дорівнює `NULL`, то функція автоматично відкриває в динамічній пам'яті буфер заданої ємності. Параметр `bmode` задає режим буферизації і може приймати одне з трьох значень:

- `_IOFBF` (у Borland C ця макроконстанта має значення `0`) – буферизація виконується так само, як у стандартних режимах;
- `_IOLBF` (або `1`) – у разі потоку виведення буфер скидається у файл кожен раз, коли в нього заноситься символ `'\n'` (рядкова буферизація); потоки введення буферизуються звичайним чином;
- `_IONBF` (або `2`) – буферизація не виконується взагалі, значення `fbuf` та `bsize` ігноруються.

Останній параметр функції `setvbuf()` – `bsize` задає ємність створеного буфера і може приймати довільне цілочислове значення, що не перевищує 64 Кбайт.

Розглянемо такі три приклади:

```
setvbuf(fp1, NULL, _IOFBF, 4096);
setvbuf(fp2, mybuffer, _IOLBF, 128);
setvbuf(fp3, nobuf, _IONBF, 0);
```

У першому прикладі для потоку `fp1` автоматично створюється буфер, ємність якого становить 4096 байтів, в усьому іншому процеси файлового обміну відбуватимуться як звичайно. У другому прикладі для потоку `fp2` буде використовуватись буфер `mybuffer`

ємністю 128 байтів, який попередньо має бути оголошений як масив або виділений у динамічній пам'яті. Обмін даними виконуватиметься через рядкову буферизацію. У третьому прикладі для потоку `fp3` буферизацію відмінено.

У разі успішного виконання функція `setvbuf()` повертає нуль, а в разі невдачі – ненульове значення (код помилки). Обидві функції керування буферизацією `setbuf()` і `setvbuf()` повинні викликатись відразу після відкриття потоку.

У середовищі Borland C за замовчуванням встановлено, що стандартний потік виведення `stdout` не здійснює буферизації. Стандартні установки потоку в разі потреби можна змінити, наприклад, зробити виведення рядково буферизованим:

```
setvbuf(stdout, NULL, _IOLBF, 80);
```

Водночас, стандартний потік введення з клавіатури `stdin` є рядково буферизованим, ємність буфера становить 128 байтів. Змінити параметри буфера введення з клавіатури не можна, оскільки процес введення базується на внутрішній буферизації MS DOS. Тому найдовший рядок, який вводиться з клавіатури за одне звертання до `gets()` чи інших функцій потокоорієнтованого введення може містити тільки 127 символів. Читання довших рядків треба організовувати програмно, виконуючи послідовні звертання до функцій буферизованого введення або застосовуючи функції консольного введення з `<conio.h>`.

15.9. Витирання та перейменування файлів

У багатьох практичних задачах потрібно редагувати або частково змінювати вміст файла. Здебільшого процес редагування файла включає таку послідовність дій:

1) відкрити для читання файл, що має бути змінений;
2) створити новий тимчасовий файл для запису;
3) у новий файл переписати вміст базового файла, виконавши потрібні зміни та доповнивши його необхідною інформацією;
4) закрити обидва файли;
5) витерти файл, який редагувався;
6) надати новоствореному файлу ім'я базового.

Конкретні реалізації задач редагування вмісту файла можуть мати свої особливості. Наприклад, базовий файл можна не витирати, а зберігати зі зміненим іменем чи розширенням (загальноприйнятою є заміна розширення на `*.bac` – *back*-версія). Деколи після завершення роботи витирають тимчасовий файл або переписують його в інший каталог тощо.

Для програмування кроків 5 та 6 наведеної схеми редагування файлів доцільно використовувати бібліотечні функції витирання та перейменування файлів, оголошені разом з іншими функціями файлового обміну в `<stdio.h>`.

Функція

```
int  remove (char* fname);
```

витирає файл з іменем `fname` (правила запису імені файла такі ж, як і для функції `fopen()`). За умови успішного витирання файла `remove()` повертає нуль, а в разі невдачі – ненульове значення.

У Borland C `remove()` організовано як макрос, який викликає функцію `unlink()`, що власне реалізує процес витирання файла:

```
int  unlink (char* fname);
```

Функція `unlink()` не підтримується стандартом мови С.

Зміну імені файла виконує функція

```
int  rename (char* oldfname, char* newfname);
```

У разі успішного виконання функція повертає нуль, а файлу, ім'я якого задається параметром `oldfname`, присвоюється нове ім'я – `newfname`.

У програмах, що реалізуються в середовищі Borland C, перейменування файлів виконується за правилами MS DOS. Важливо, щоб на момент перейменування у каталозі не було іншого файла з іменем `newfname`, бо тоді ім'я файла не змінюється, а функція повертає значення, що не дорівнює нулю. Тому в процесі редагування файла спочатку витирають або перейменовують базовий файл, а вже потім його ім'я надають новоствореному.

Всі функції витирання та перейменування файлів працюють тільки зі закритими файлами.

Створення нових і тимчасових файлів пов'язане з необхідністю перевірки, чи існує інший файл з іменем, яке повинен отримати новостворений файл. Якщо такий контроль не здійснювати, то можна випадково затерти наявний однойменний файл. Альтернативним вирішенням даної проблеми є використання функції, що генерує ім'я файла, яке буде унікальним для поточного активного каталога:

```
char*  tmpnam (char* fname);
```

Єдиним параметром `tmpnam()` є вказівник на масив символів, в який буде записано згенероване ім'я. Розмірність масиву `fname` повинна бути не меншою за значення макроконстанти `L_tmpnam` (13 символів). Якщо замість `fname` підставити значення `NULL`, то згенероване функцією ім'я буде записане у спеціальну внутрішню змінну, адресу якої поверне функція. У Borland C унікальні імена створюються за таким правилом: якщо в активному каталозі відсутній файл з іменем `tmp1.$$$`, то це буде першим згенерованим ім'ям, інакше послідовно перевіряються імена `tmp2.$$$`, `tmp3.$$$` і т.д. Кожен виклик функції `tmpnam()` у програмі генерує наступне за порядковим номером ім'я файла.

У разі успішного виконання `tmpnam()` повертає адресу першого символа згенерованого імені файла. Якщо ж унікальне ім'я згенерувати не вдалось, то функція повертає `NULL`.

Проілюструємо застосування описаних вище функцій прикладом програми, яка виконує редагування вмісту файла. Програма реалізує таку задачу: в заданому текстовому файлі треба доповнити кожну цифру словесним найменуванням (тобто відповідним чисельником), записаним у круглих дужках. Відредагованому файлу треба надати ім'я, яке мав базовий файл.

```
/*******************************************************/
/*  Доповнення файла найменуваннями цифрових символів  */
/*******************************************************/

#include <stdio.h>
#include <ctype.h>
int main(void)
{
    FILE *fold, *ftmp;          /* вказівники базового та нового файлів */
    char foldname[13], ftmpname[13];
    int symb;
    char *number[] = {"нуль", "один", "два", "три", "чотири",
                      "п\'ять", "шість", "сім", "вісім", "дев\'ять"};
  printf ("Файл для заміни - ");
  gets(foldname);
  if ( (fold = fopen(foldname,"rt"))== NULL ) {
     printf ("Помилка в імені файла - %s \n", foldname);
     return 0;
  }
  tmpnam(ftmpname);              /* генерування унікального імені файла */
  if ((ftmp = fopen(ftmpname,"wt"))== NULL ) {
     printf ("Не можна створити новий файл");
     return 0;
  }
  while ((symb = getc(fold))!= EOF ) {        /* цикл редагування файла */
     putc(symb, ftmp);
     if ( isdigit(symb) )        /* якщо зустрілась цифра, то виводиться */
        fprintf (ftmp, "(%s)", number[symb-'0']);        /* її назва */
  }
  fcloseall();                                          /* закриття файлів */
  if (remove(foldname)== 0) {             /* витирання початкового файла */
     rename(ftmpname, foldname);          /* перейменування нового файла */
     printf("Файл %s відредаговано.\n", foldname);
  } else
     printf("Створено новий файл - %s \n", ftmpname);
    return 0;
}
```

Файл, призначений для тимчасового використання, можна створити за допомогою функції

```
FILE*  tmpfile (void);
```

Вона відкриває тимчасовий файл у режимі бінарного читання/запису (`"w+b"`). Файл отримує унікальне ім'я згідно з `tmpnam()`. Треба пам'ятати, що створюється тимчасовий файл – він буде автоматично витертий після закриття чи завершення роботи програми. Функція `tmpfile()` повертає вказівник на потік, пов'язаний зі створеним тимчасовим файлом, або `NULL`, якщо файл відкрити не вдалось.

15.10. Інші засоби для роботи з файлами

15.10.1. Функції <dir.h> для роботи з каталогами та файлами

Додаткові можливості щодо роботи з файлами та каталогами MS DOS надають програмістові бібліотечні функції Borland C, прототипи яких оголошені в заголовному файлі <dir.h>. Серед них:

- `getdisk()/setdisk()` – визначення/зміна поточного системного диска;
- `mkdir()/chdir()/rmdir()` – створення/зміна/витирання каталога;
- `getcurdir()/getcwd()` – визначення активного каталога для поточного або заданого диска;
- `mktemp()` – генерування унікального імені файла;
- `findfirst()/findnext()` – пошук у заданому каталозі першого/наступного файла, ім'я якого відповідає вказаному шаблону;
- `fnsplit()` – формування повного імені файла зі складових частин;
- `searchpath()` – визначення шляху до файла.

Детальнішу інформацію про параметри, повернене значення та особливості застосування цих функцій можна знайти в [8] або отримати з підсистеми допомоги Help інтегрованого середовища Borland C.

15.10.2. Низькорівневе звертання до файлів

Як вже зазначалось на початку розділу, функції низькорівневого файлового обміну даними є системно і апаратно залежними, тому вони не підтримуються стандартом мови С. Операції введення/виведення даних на низькому рівні в середовищі Borland C базуються на засобах файлового обміну MS DOS. Ці операції блокоорієнтовані, не використовують проміжної буферизації і не здійснюють форматування даних. Зв'язок із файлом, відкритим на низькому рівні, реалізується через т. зв. *дескриптор* (*handle*) файла, який в літературі також називають *префіксом*. Дескриптор файла – це цілочислова змінна, в яку після відкриття файла заноситься номер, за яким цей файл зареєстровано у внутрішніх таблицях операційної системи, де зберігається інформація про відкриті файли. Всі функції опрацювання файла, зокрема ті, що виконують введення/виведення даних, звертаються до файла через його дескриптор.

Прототипи функцій, які реалізують роботу з файлами на низькому рівні, записані в заголовному файлі <io.h>. Основними серед них є:

- `open()/creat()` – відкрити/створити файл;
- `sopen()` – відкрити файл для спільного використання декількома процесами;
- `read()/write()` – читання/запис даних;
- `lseek()` – позиціювання покажчика поточної позиції файла;
- `tell()` – визначення поточної позиції файла;
- `eof()` – перевірка досягнення кінця файла;
- `close()` – закриття файла.

Ми не будемо розглядати перелічені функції низькорівневого файлового обміну даними. Відповідну інформацію можна знайти в [3, 8, 15] або отримати з підсистеми допомоги Help інтегрованого середовища Borland C.

Якщо файл відкрито для потокоорієнтованого обміну, то значення дескриптора можна отримати через функцію

```
int  fileno (FILE* fp);
```

оголошену в <stdio.h>. Функція повертає дескриптор файла, пов'язаного з потоком fp. Це дає змогу додатково до функцій високорівневого обміну даними застосовувати до файла з потоку fp необхідні функції з <io.h>. Зокрема, корисною може бути функція

```
long  filelength (int handle);
```

яка визначає розмір (у байтах) відкритого файла, заданого дескриптором handle. У разі виникнення помилки функція повертає значення -1L.

Раніше для визначення розміру файла ми розробили окрему функцію FileSize(), в якій використали декілька стандартних бібліотечних функцій: fseek(), ftell(), fgetpos() та fsetpos() (див. параграф 15.6). Через filelength() розмір файла, пов'язаного з потоком fp, можна знайти так:

```
/*  Визначення розміру файла  */
#include <stdio.h>
#include <io.h>
  FILE *fp;
  long fsize;
. . .                              /* відкриття файла функцією fopen() */
fsize = filelength( fileno(fp) );
```

Корисною для багатьох практичних задач є також функція

```
int  chsize (int handle, long newsize);
```

яка змінює розмір відкритого файла на новий, заданий параметром newsize. Якщо значення newsize менше за поточний розмір файла, то всі кінцеві байти втрачаються (файл обтинається), інакше файл доповнюється нульовими байтами до потрібної довжини. Наприклад, виклик

```
chsize (fileno(fp), 250L);
```

обмежить розмір файла, пов'язаного з потоком fp, 250-ма початковими байтами. Функція chsize() повертає нуль у разі успішного виконання та -1 у разі невдачі.

Дві наступні функції:

```
int  getftime (int handle, struct ftime* timebuf);
int  setftime (int handle, struct ftime* timebuf);
```

дають змогу отримати/встановити дату і час системної реєстрації файла, заданого

дескриптором handle. Дані заносяться в змінну (чи ділянку пам'яті) або відповідно зчитуються зі змінної (чи ділянки пам'яті), на яку вказує timebuf. Змінна, адресу якої задає параметр timebuf, повинна бути структурою, точніше – полем бітів зі шаблоном struct ftime, оголошеним в <io.h>. Шість полів цієї структури призначені для збереження року, місяця, дня, години, хвилин і секунд, що характеризують часові параметри файла. Функція повертає нуль у разі успішного виконання та значення -1, якщо виникає помилка.

Інформацію про відкритий файл можна отримати також за допомогою функції fstat(), оголошеної в <sys\stat.h>:

```
int  fstat (int handle, struct stat* statbuf);
```

Перший параметр функції fstat() – handle задає дескриптор файла, а другий – statbuf вказує адресу змінної (ділянки пам'яті), куди будуть записані дані. Відповідна змінна повинна бути структурою, що має встановлений шаблон struct stat. У разі успішного виконання функції в поля цієї структури заноситься інформація про файл, а саме: диск, на якому записаний файл, режими відкриття файла і доступу до нього, розмір файла в байтах, час останньої модифікації файла тощо. Функція повертає нуль у разі нормального завершення та -1, якщо зафіксовано помилку.

Запитання та завдання для самоконтролю

1. Функції обміну даними з файлами та консольними пристроями поділяють на три рівні – назвіть їх. У яких заголовних файлах записані прототипи функцій кожного з цих рівнів?
2. Чим відрізняється високорівневий потокоорієнтований обмін даними від низькорівневого файлового обміну?
3. Що називають потоком даних? Яке призначення проміжної буферизації? Коли відбувається скидання буфера виведення у файл?
4. На які групи можна поділити стандартні бібліотечні функції <stdio.h>?
5. Яка функція відкриває потік для файлового обміну? Що є її параметрами? Який тип має значення, яке повертає ця функція?
6. Назвіть три стандартні потоки, що автоматично відкриваються С-програмами. Коли доцільно робити перескерування потоків?
7. Які функції використовують для посимвольного зчитування текстових файлів і посимвольного запису інформації у файл?
8. У чому відмінність функцій fgets() і gets() та функцій fputs() і puts()? Як кожна з цих функцій опрацьовує символи кінця рядка?
9. Напишіть виклик функції fwrite(), який дає змогу занести в бінарний файл, заданий вказівником fp, блок із 200 довгих цілих чисел, що зберігаються у масиві resdat.
10. За допомогою яких функцій можна виконувати форматне зчитування вхідних даних із текстового файла і запис у файл результатів роботи програми?

11. Які дії виконує кожна з наступних функцій: `fseek()`, `rewind()`, `fgetpos()`, `fsetpos()`, `ftell()`?

12. У яких випадках треба застосовувати функцію `feof()`? Яке значення повертає ця функція? Якими функціями можна скинути індикатор кінця файла?

13. Коли потрібно очищати буфери потоків введення і потоків виведення? Як це зробити?

14. Яку послідовність дій треба виконати, щоб змінити вміст заданого файла?

15. Що повертає функція `fileno()`? Як за допомогою цієї функції визначити розмір відкритого файла?

Запрограмуйте наведені нижче задачі, які вимагають опрацювання і/або створення файлів різних типів

16. У заданому з командного рядка файлі з текстом С-програми порахувати загальну кількість операторів циклу (*підказка*: оператори циклу визначати за відповідними службовими словами).

17. Створити бінарний файл rndnumb.dat, який заповнити двійковими кодами `N` випадкових чотирицифрових парних чисел.

18. Задано два текстові файли. Перевірити, чи їх вміст збігається. Якщо так, то витерти другий файл, у противному разі вивести пари рядків, у яких виявлено відмінності.

19. У текстовий файл pascal.trg послідовно записати `K` початкових рядків трикутника Паскаля. Кожен елемент (крім крайніх) у трикутнику Паскаля дорівнює сумі двох сусідніх елементів, розташованих над ним у попередньому рядку. Перші шість рядків трикутника Паскаля такі:

```
     1
    1 1
   1 2 1
  1 3 3 1
 1 4 6 4 1
1 5 10 10 5 1
```

20. Задано файл з текстом Pascal-програми. Опрацювати текст програми, вилучивши з нього всі коментарі. Новий текст повинен зберігатись у файлі з тим самим іменем, а попередня версія – у файлі з розширенням `*.bac`.

21. У файлі newvers.txt записано текст українською мовою. Визначити середній розмір слів файла в літерах (розділові знаки та інші небуквенні символи не враховувати). Одночасно вивести на екран усі слова з цього файла, які складаються із десяти та більше літер. У програмі використати окрему функцію, яка перевіряє, чи даний символ є літерою української абетки.

Розділ 16

КОНСОЛЬНИЙ ОБМІН ДАНИМИ

У цьому розділі:

- Загальна характеристика функцій консольного обміну даними
- Перелік функцій `<conio.h>`, призначених для керування відображенням інформації в текстових режимах роботи відеосистеми
- Встановлення атрибутів символів і режиму блимання
- Робота з текстовими вікнами: активізація вікна, побудова рамки вікна, зчитування параметрів активного вікна, копіювання та відновлення вікон
- Функції консольного виведення тексту та редагування екранного зображення
- Керування позицією та формою текстового курсора
- Короткий опис процесу введення даних з клавіатури
- Групи клавіш; розширені ASCII-коди клавіш керування, функціональних клавіш і клавішних комбінацій
- Особливості функцій консольного введення даних; функція перевірки буфера клавіатури `kbhit()`
- Аналіз стану перемикальних і шифт-клавіш

Переважна більшість програм вимагає введення певних даних з клавіатури та відображення текстових повідомлень і результатів виконання програми на екрані монітора. Такий обмін інформацією з користувачем називають *консольним* введенням/виведенням даних. У попередньому розділі ми розглядали стандартні бібліотечні функції мови С, оголошені в `<stdio.h>`, що реалізують високорівневий потокоорієнтований обмін даними з консольними пристроями. Ці функції універсальні й мобільні, але вони обмежені за своїми можливостями, зокрема, не дають змоги змінювати колір літер, створювати текстові вікна, користуватись клавішами стрілок тощо.

Бібліотека системи програмування Borland C містить набір спеціальних функцій консольного введення/виведення даних, які характеризуються достатньо високою швидкодією і надають користувачеві додаткові можливості щодо керування формою відображення інформації на екрані та способом введення даних з клавіатури. Функції

консольного обміну даними не належать до стандартизованих, оскільки вони базуються на конкретному апаратно-програмному забезпеченні комп'ютерів. Прототипи всіх функцій консольного введення/виведення оголошені в заголовному файлі `<conio.h>`.

16.1. Керування консольним виведенням текстової інформації

Повний перелік функцій `<conio.h>`, призначених для керування процесом відображення текстових повідомлень і даних, наведено в табл. 16.1.

Всі функції консольного виведення можна поділити на шість груп:

1) функція встановлення заданого текстового режиму: `textmode();`
2) функції встановлення атрибутів символів: `textcolor()`, `textbackground()`, `textattr()`, `highvideo()`, `lowvideo()`, `normvideo();`

Таблиця 16.1

Функції керування консольним виведенням інформації

Найменування функції	Призначення
`clreol()`	Очищення рядка, починаючи з поточної позиції курсора
`clrscr()`	Очищення активного вікна екрана
`cprintf()`	Форматне виведення даних
`cputs()`	Виведення символьного рядка
`delline()`	Видалення рядка, на якому знаходиться курсор
`gettext()`	Запис у зовнішній буфер вмісту заданого текстового вікна
`gettextinfo()`	Визначення параметрів активного текстового вікна
`gotoxy()`	Встановлення курсора у задану позицію текстового вікна
`highvideo()`	Встановлення режиму підвищеної яскравості символів
`insline()`	Вставлення порожнього рядка в позицію, задану курсором
`lowvideo()`	Встановлення режиму зменшеної яскравості символів
`movetext()`	Копіювання текстового вікна в задане місце екрана
`normvideo()`	Відновлення початкових кольорів символів і фону
`putch()`	Виведення одного символа
`puttext()`	Відновлення текстового вікна, скопійованого у буфер
`_setcursortype()`	Встановлення форми текстового курсора
`textattr()`	Встановлення атрибутів символів
`textbackground()`	Встановлення кольору фону символів
`textcolor()`	Встановлення кольору зображення символів
`textmode()`	Встановлення заданого текстового режиму
`wherex()`	Визначення горизонтальної координати курсора
`wherey()`	Визначення вертикальної координати курсора
`window()`	Створення текстового вікна за заданими координатами

3) функції формування текстових вікон та роботи з вікнами: `window()`, `clrscr()`, `gettextinfo()`, `gettext()`, `puttext()`, `movetext()`;
4) функції виведення символів, рядків і числових даних: `putch()`, `cputs()`, `cprintf()`;
5) функції редагування тексту у вікні екрана: `clreol()`, `delline()`, `insline()`;
6) функції керування позицією та формою текстового курсора: `gotoxy()`, `wherex()`, `wherey()`, `_setcursortype()`.

Стандартні текстові режими. Стандартні текстові режими відеоадаптерів VGA та SVGA мають визначені розміри екранного зображення:

- 25 рядків по 40 символів у рядку – режими 0 та 1;
- 25 рядків по 80 символів у рядку – режими 2, 3 та 7;
- 50 рядків по 80 символів у рядку – режим 64.

Усі перелічені режими, крім 7-го, кольорові та дають змогу відтворювати символи з використанням 16-колірної палітри. Режим із номером 7 монохромний. Для відеоадаптерів VGA/SVGA режими 0 і 1 та режими 2 і 3 однакові.

Основним текстовим режимом, який автоматично встановлюється відеосистемою, є режим номер 3. Змінити текстовий режим на інший можна за допомогою функції

```
void textmode (int tmode);
```

тут `tmode` – номер текстового режиму, що має бути встановлений. Параметр `tmode` можна задати числовим значенням номера режиму або відповідною іменованою константою з переліку `text_modes`, оголошеного в `<conio.h>`: `BW40` (режим 0), `C40` (режим 1), `BW80` (режим 2), `C80` (режим 3), `MONO` (режим 7), `C4350` (режим 64). Якщо викликати `textmode()` з параметром `-1` (або відповідною іменованою константою `LASTMODE`), то відбувається перевстановлення поточного режиму.

Кожна зміна текстового режиму на інший (крім випадків встановлення режиму з номером 64), пов'язана з очищенням відеопам'яті – попереднє екранне зображення витирається, встановлюється повноекранне вікно, відновлюються атрибути символів, які були встановлені на момент запуску програми, курсор переводиться у позицію (1, 1). У разі повторного задання режиму з тим самим номером, зокрема через параметр `LASTMODE` (або `-1`), кольорове екранне зображення та поточна позиція курсора зберігаються, але встановлюється повноекранне вікно та відновлюються атрибути символів, які зафіксовані у відеосистемі як початкові. Встановлені параметри будуть впливати на наступні операції консольного виведення.

16.1.1. Встановлення атрибутів символів

Екранне зображення зберігається у виділеній області оперативної пам'яті, яку називають *відеопам'яттю*. Кожен символ екрана (знакомісце) займає у відеопам'яті два байти: у молодший байт заноситься ASCII-код символа, а в старший – атрибути даного символа (див. рис. 14.3). Структуру байта атрибутів і призначення кожного біта цього байта показано на рис. 16.1. Біти 0..3 задають колір відтворення символа,

а біти 4..6 – колір фону, на якому зображується цей символ (B/b – синя, G/g – зелена, R/r – червона складова кольору). Найстарший 7-й біт відіграє подвійну роль: якщо у відеосистемі включено режим блимання символів, то запис 1 у цей біт викличе періодичне блимання символа; якщо ж режим блимання відключено, то 7-й біт байта атрибутів задає підвищену яскравість кольору фону, так само, як 3-й біт задає яскравість кольору символа. Стандартно відеосистема встановлює світло-сірий колір символів і чорний колір фону, що відповідає двійковому коду байта атрибутів 00000111 або шістнадцятковому значенню 0x07. Коли, наприклад, надати байту атрибутів шістнадцяткове значення 0xae (його двійковий код 10101110), то відповідний символ буде відображений жовтим кольором на зеленому фоні та блиматиме (за умови режиму блимання).

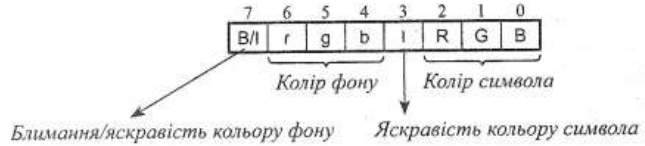


Рис. 16.1. Структура байта атрибутів

Бібліотечні функції консольного виведення дають змогу встановлювати значення цілого байта атрибутів або змінювати окремо колір символа чи колір фону. Кольори можуть задаватися як відповідним числовим значенням, так і за допомогою іменованих констант, значення яких оголошені в <conio.h> через перелік COLORS:

```
enum COLORS {
    BLACK,              /*  0 - чорний  */
    BLUE,               /*  1 - синій  */
    GREEN,              /*  2 - зелений  */
    CYAN,               /*  3 - морської хвилі  */
    RED,                /*  4 - червоний  */
    MAGENTA,            /*  5 - фіолетовий  */
    BROWN,              /*  6 - коричневий  */
    LIGHTGRAY,          /*  7 - світло-сірий  */
    DARKGRAY,           /*  8 - темно-сірий  */
    LIGHTBLUE,          /*  9 - світло-синій  */
    LIGHTGREEN,         /*  10 - яскраво-зелений  */
    LIGHTCYAN,          /*  11 - блакитний  */
    LIGHTRED,           /*  12 - світло-червоний  */
    LIGHTMAGENTA,       /*  13 - малиновий  */
    YELLOW,             /*  14 - жовтий  */
    WHITE,              /*  15 - білий  */
};
```

Палітра кольорів, які реально висвітлюються на екрані, залежить від поточних характеристик конкретної відеосистеми, тому гарантувати ідентичність зображення, перенесеного на інший комп'ютер, не можна.

Зміну кольору символів здійснює функція

```
void textcolor (int color);
```

Значення параметра `color` може бути задане іменованою константою з переліку `COLORS` або відповідним числовим значенням. Всі наступні операції консольного виведення даних будуть здійснюватися цим кольором. Якщо до значення кольору додати макроконстанту `BLINK` (її значення дорівнює `128`), то в біт блимання (біт 7 байта атрибутів) потрапляє одиниця, і виведені на екран символи блиматимуть, якщо у відеосистемі включено режим блимання, інакше символи будуть відтворюватися з підвищеною яскравістю фону.

Призначення 7-го біта байтів атрибутів можна встановити через функцію 10h (підфункція 03h) BIOS-переривання з номером 10h. Записана нижче функція `Blink` включає режим блимання, якщо її вхідний параметр `blk` дорівнює `1`, або режим підвищеної яскравості фону, якщо значення `blk` дорівнює `0`. Виконання функції вплине на відображення всіх символів екрана, 7-й біт байта атрибутів яких встановлено в одиницю.

```
/*  Функція перемикання режиму блимання  */
void Blink (int blk)
{
   _AH = 0x10;            /* номер функції BIOS-переривання */
   _AL = 0x03;            /* номер підфункції даного переривання */
   _BL = blk;
   geninterrupt (0x10); /* виклик переривання 10h */
}
```

Функція

```
void textbackground (int bkcolor);
```

встановлює колір фону символів, заданий параметром `bkcolor`. Усі наступні символи будуть виводитись на екран зі заданим кольором фону. Оскільки для кольору фону в байті атрибутів виділено три біти (див. рис. 16.1), то стандартно палітра кольорів фону обмежена значеннями від `0` до `7`.

Змінити значення всього байта атрибутів (тобто колір символів і колір фону одночасно) можна за допомогою функції

```
void textattr (int sattr);
```

Параметр `sattr` задає нове значення атрибутів символа, яке буде використовуватись у всіх наступних функціях консольного виведення. Атрибути можна задавати числовим значенням, або формувати з іменованих констант переліку `COLORS`. Наприклад, якщо байт атрибутів задати так:

```
textattr (BLUE << 4 | YELLOW);          /* або  textattr(0x1e) */
```

то надалі на екран виводитимуться жовті символи на синьому фоні. Щоб забезпечити блимання символів, треба додати макроконстанту `BLINK` (або `128`):

```
textattr (YELLOW | BLUE << 4 | BLINK);    /* або  textattr(0x9e) */
```

Яскравістю та кольором символів керують також наступні три функції:

```
void  highvideo (void);
void  lowvideo (void);
void  normvideo (void);
```

які задають відповідно підвищену, знижену та початкову яскравість для символів, що будуть виводитись на екран. Функція `highvideo()` встановлює в 1 біт яскравості кольору символа в байті атрибутів (див. рис. 16.1), а функція `lowvideo()` скидає цей біт у 0. Функція `normvideo()` відновлює ті кольори символів і фону, які були встановлені на момент запуску програми.

16.1.2. Формування текстових вікон

Функції даної групи дають змогу виділяти на екрані активне текстове вікно та здійснювати операції над вікнами. *Активним вікном* називають прямокутну область екрана, в межах якої здійснюється поточне виведення текстових повідомлень. Стандартно встановлюється повноекранне вікно.

Створення вікна. Для активізації текстового вікна призначена функція

```
void  window (int left, int top, int right, int bot);
```

Параметри функції задають координати вікна: перша пара аргументів встановлює позицію лівого верхнього кута (`left` – горизонтальна, `top` – вертикальна координати), а друга пара задає знакомісце правого нижнього кута вікна (`right` – горизонтальна, `bot` – вертикальна координати). Координати вказуються відносно початку екрана (*екранна система координат*), рядки та стовпчики нумеруються, починаючи з 1 (рис. 16.2).

У разі успішного відкриття вікна курсор встановлюється в позицію його лівого верхнього кута. Ця позиція отримує координати (1, 1) – відносно неї тепер будуть відраховуватись всі інші координати (*віконна система координат*). Наступні операції консольного виведення відображатимуть інформацію в межах встановленого вікна. У разі досягнення правої межі вікна курсор автоматично переводиться на початок

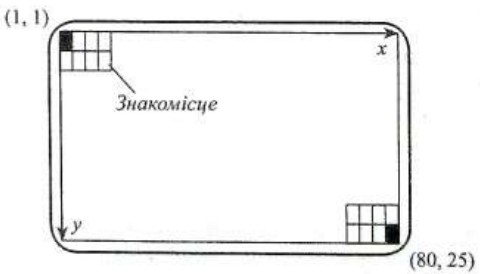


Рис. 16.2. Система екранних координат текстового режиму номер 3

наступного рядка, а в разі досягнення курсором правої нижньої позиції вікна відбувається автоматичний *скролінг* зображення: текст вікна піднімається на один рядок угору, нижній рядок вікна звільняється і заповнюється поточним кольором фону, верхній рядок вікна втрачається.

Іноді виникає потреба тимчасово відмінити скролінг, щоб можна було вивести символ в останній позиції вікна. В `<conio.h>` оголошено спеціальну внутрішню змінну `_wscroll`, що має тип `int`, через яку можна керувати автоматичним скролінгом. Якщо змінній `_wscroll` присвоїти значення `0`, то автоматичний скролінг скасовується. Щоб відновити скролінг вікна, достатньо у змінну `_wscroll` занести значення `1`.

Скасування скролінгу автоматично скасовує переведення курсора на новий рядок. У разі досягнення правої межі вікна виведення даних продовжується від початку поточного рядка. Тому слід відновлювати скролінг відразу після заповнення нижньої частини вікна.

Відкриття вікна не очищує область екрана, зайняту цим вікном. Щоб звільнити вікно від попередніх даних і зафарбувати його поточним кольором фону, використовують функцію

```
void  clrscr (void);
```

Для заповнення вікна `clrscr()` застосовує колір фону, що був встановлений останнім викликом функції `textbackground()` або `textattr()`.

Вікна з рамками. Якщо потрібно створити текстове вікно, виділене рамкою, то найкраще застосувати таку послідовність дій:

1) вивести рамку вікна в повноекранному режимі (це дасть змогу уникнути скролінгу після виведення символа правого нижнього кута рамки);
2) відкрити вікно всередині рамки.

Для формування рамки вікна застосовують спеціальні символи псевдографіки, коди яких наведено в табл. 16.2. Рамку можна зображати символами, які складаються з одинарних або подвійних ліній. Можна також будувати комбіновані рамки, використовуючи відповідні символи (див. табл. Д1.2 в Додатку 1).

Таблиця 16.2

Символи псевдографіки, що застосовуються для побудов рамок вікон

Символ	Код	Символ	Код	Символ	Код	Символ	Код
│	179	┘	217	‖	186	╜	188
─	196	┤	180	═	205	╡	181
┌	218	├	195	╒	201	╞	198
┐	191	┴	193	╖	187	╨	208
└	192	┬	194	╚	200	╥	210

Для введення з клавіатури символів псевдографіки можна скористатись таким прийомом: натиснувши клавішу *Alt*, набрати на цифровій клавіатурі (спеціальна права частина клавіатури) код даного символа. Після відпускання *Alt* на екрані відобразиться потрібний символ.

Наведемо функцію OpenWindow(), яка формує текстове вікно з подвійною рамкою та заголовним надписом.

```
/*  Функція створення текстового вікна з рамкою і заголовком  */
# include <string.h>
# include <conio.h>
void  OpenWindow (int x1, int y1, int x2, int y2,
                  int wcol, int wbk, char* title)
/***          Параметри функції:
       x1, y1, x2, y2 - координати двох протилежних кутів вікна;
       wcol, wbk - колір символів рамки вікна та колір фону;
       title - заголовок вікна                                    ***/
{
    int x,y;
  window(1, 1, 80, 25);          /* встановлення повноекранного вікна */
  textcolor(wcol);                    /* задання кольору символів та */
  textbackground(wbk);                                 /* кольору фону */
  gotoxy(x1, y1);
  putch(201);                                          /* символ '╔' */
  for (x=x1+1;  x<x2;  x++)           /* виведення верхньої лінії */
    putch(205);                                        /* символ '═' */
  putch(187);                                          /* символ '╗' */
  for (y=y1+1;  y<y2;  y++) {         /* виведення бокових ліній  */
    gotoxy(x1, y);
    putch(186);                                        /* символ '║' */
    gotoxy(x2, y);
    putch(186);
  }
  gotoxy(x1, y2);
  putch(200);                                          /* символ '╚' */
  for (x=x1+1;  x<x2;  x++)           /* виведення  нижньої лінії */
     putch(205);                                       /* символ '═' */
  putch(188);                                          /* символ '╝' */
  if (*title!=0) {                      /* задано заголовок вікна */
     x=(x2-x1+1-(strlen(title)+2))/2;          /* позиція заголовка */
     if (x>1) {                          /* заголовок вміщається */
        gotoxy(x1+x+1, y1);
        putch(181);              /* символ лівого краю заголовка '╡' */
        cputs(title);                              /* текст заголовка */
        putch(198);              /* символ правого краю заголовка '╞' */
     }
  }
  window(x1+1, y1+1, x2-1, y2-1);              /* вікно всередині рамки */
  clrscr();                                      /* очищення вікна */
}
```

Приклад звертання до функції відкриття вікна з рамкою:

```
OpenWindow (25,8, 55,20, WHITE, BROWN, "Результати");
```

Побудову текстового вікна з тінню доцільно реалізовувати через наступну послідовність дій:

1) за допомогою функції `window()` відкрити у повноекранному режимі вікно тіні зміщене вправо вгору (або вниз) відносно основного вікна;
2) встановити потрібний колір фону (колір тіні) та зафарбувати створене вікно, використовуючи функції `textbackground()` та `clrscr()`;
3) знову перейти в повноекранний режим і створити основне вікно (якщо потрібно, то з рамкою) за допомогою дій, описаних вище;
4) встановити колір фону та колір символів для основного вікна й очистити його.

Зчитування параметрів активного вікна. Інформацію про параметри активного текстового вікна та загальні характеристики встановленого відеорежиму можна отримати через функцію

```
void gettextinfo (struct text_info * winf);
```

Ця функція має один параметр `winf`, який вказує на структуру зі шаблоном `struct text_info`, оголошеним у `<conio.h>` наступним чином:

```
struct text_info {
  unsigned char winleft;              /* координата лівої межі вікна */
  unsigned char wintop;              /* координата верхньої межі вікна */
  unsigned char winright;             /* координата правої межі вікна */
  unsigned char winbottom;           /* координата нижньої межі вікна */
  unsigned char attribute;          /* атрибути тексту активного вікна */
  unsigned char normattr;               /* початкові атрибути тексту */
  unsigned char currmode;            /* встановлений текстовий режим */
  unsigned char screenheight;          /* вертикальний розмір екрана */
  unsigned char screenwidth;         /* горизонтальний розмір екрана */
  unsigned char curx;     /* поточна горизонтальна координата курсора */
  unsigned char cury;       /* поточна вертикальна координата курсора */
);
```

У результаті виконання функції `gettextinfo()` поля структури, на яку вказує `inf`, заповнюються даними про поточні параметри активного вікна: кооординати вікна, атрибути символів (встановлені й базові), позицію текстового курсора, а також даними про загальні відеохарактеристики: текстовий режим і розміри екрана.

Застосовуючи функцію `gettextinfo()`, можна створювати універсальні користувацькі функції для роботи з текстовими вікнами. Як приклад запишемо функцію `OutFinalMsg()`, призначену для виведення заданого короткого повідомлення в правій частині останнього рядка активного текстового вікна. Повідомлення повинно виводитись кольором, інверсним до кольору фону вікна.

```
/* Функція виведення повідомлення в правій нижній частині вікна */
#include <conio.h>
#include <string.h>
void OutFinalMsg (char* msg)
{
    int x, y;
    struct text_info winf;        /* змінна для запису відеопараметрів */
   gettextinfo(&winf);          /* визначення параметрів активного вікна */
   y=winf.winbottom-winf.wintop+1; /* номер останнього рядка вікна */
   x=winf.winright-winf.winleft-strlen(msg)+2;     /* горизонтальна */
   gotoxy(x,y);                          /* позиція початку повідомлення */
   textcolor((~winf.attribute&0xf0)>> 4);   /* встановлення кольору:
 інвертуємо байт атрибутів, виділяємо колір фону і зсуваємо його вправо */
  _wscroll=0;                                  /* скасування скролінгу  */
  cputs(msg);                          /* виведення заданого повідомлення */
  _wscroll=1;                                  /* відновлення скролінгу */
}
```

Приклад звертання до функції:

```
OutFinalMsg ("ESC - завершення роботи");
```

Копіювання та відновлення вікон. Три наступні бібліотечні функції призначені для збереження, відновлення та копіювання текстових вікон, зокрема повноекранних.

Функція

```
int  gettext (int left, int top, int right, int bot, void* buf);
```

послідовно записує у буфер, адресу якого задає вказівник `buf`, символи та атрибути прямокутної області екрана, заданої координатами лівого верхнього (`left`, `top`) та правого нижнього (`right`, `bot`) кутів. Координати області копіювання повноекранні. Буфер, на який вказує `buf`, має бути попередньо зарезервований як масив або як ділянка в динамічній пам'яті. Необхідна мінімальна ємність буфера становить (`right-left+1`)×(`bot-top+1`)×2 байт. У разі успішного копіювання заданого вікна у буфер функція повертає одиницю, а в разі помилки копіювання – нуль.

Наведена нижче функція `SaveWindow()` копіює текстове вікно заданого розміру в динамічну пам'ять і повертає адресу ділянки скопійованого вікна.

```
/*  Функція копіювання вікна в динамічну пам'ять  */
#include <stdlib.h>
#include <conio.h>
unsigned*  SaveWindow (int left, int top, int winw, int winh)
  /*          Параметри функції:
      left, top - координати лівого верхнього кута вікна;
      winw, winh - розміри вікна: winw - ширина, winh - висота  */
{
```

```
    unsigned *buf;
    unsigned size=winw*winh*2;              /* розмір вікна в байтах */
  buf=(unsigned*)malloc(size);          /* виділення динамічної пам'яті */
  if (!buf) return NULL;                          /* пам'ять відсутня */
  gettext(left,top,left+winw-1,top+winh-1,buf);   /* копіювання вікна */
  return buf;
}
```

Наступний фрагмент програми записує в динамічну пам'ять зображення текстового вікна з рамкою, створеного попереднім викликом функції `OpenWindow()`.

```
    unsigned *mem;
    int  w=55-25+1,   h=20-8+1;              /* розміри вікна */
  mem=SaveWindow(25,8,w,h);                  /* копіювання вікна */
```

Друга функція:

```
int  puttext (int left, int top, int right, int bot, void* buf);
```

відтворює на екрані текстове зображення з буфера, адресу початку якого задає вказівник `buf`. Вміст буфера виводиться у прямокутну область, задану координатами лівого верхнього (`left, top`) та правого нижнього (`right, bot`) кутів. Нумерація координат повноекранна. Для відображення символів використовуються значення атрибутів, записані в `buf`. Ця функція дає змогу ефективно відновлювати текстові вікна, попередньо збережені функцією `gettext()`. За умови, що координати нового вікна задано коректно та успішно відбулось виведення тексту, функція повертає ненульове значення, а в разі помилки вікно не виводиться і `puttext()` повертає нуль.

Наступний фрагмент програми відновлює на екрані та робить активним текстове вікно з рамкою, скопійоване в динамічну пам'ять у попередньому прикладі. Відновлене вікно буде розташоване в нижній правій частині екрана.

```
puttext (80-w+1, 25-h+1, 80, 25, mem);   /* відтворення вікна з рамкою */
window (80-w+2, 25-h+2, 79, 24);         /* активізація вікна */
```

Хоча правий нижній кут рамки відтвореного вікна потрапляє на знакомісце з координатами (80,25), скролінг не відбувається.

Відтворення текстового вікна пов'язане з двома особливостями, про які треба обов'язково пам'ятати. По-перше, розміри вікна, скопійованого в `buf`, і розміри прямокутної області, в якій воно відтворюється, можуть бути різними. Якщо нове вікно менше за розміром, ніж вікно, яке занесене в `buf`, то виводяться тільки ті початкові символи з числа збережених, що вміщуються у новому вікні. Здебільшого в таких випадках розташування відновленого тексту не збігатиметься зі збереженим. Якщо ж нове вікно більше за те, що відтворюється, то зайва частина нового вікна заповнюється "сміттям". По-друге, відтворення зображення вікна не робить це вікно активним, всі поточні параметри відеосистеми, зокрема позиція текстового курсора, залишаються незмінними. Щоб активізувати відтворене вікно, треба скористатись функцією `window()`.

Третя функція:

```
int  movetext (int left, int top, int right, int bot,
               int newleft, int newtop);
```

копіює вікно, задане координатами лівого верхнього (`left, top`) та правого нижнього (`right, bot`) кутів, в інше місце екрана, координати лівого верхнього кута якого задають два останні параметри (`newleft, newtop`). Базове вікно зберігається на екрані й залишається активним, поточна позиція текстового курсора не змінюється. Розміри скопійованого вікна по горизонталі та вертикалі збігаються з розмірами базового. Усі координати повноекранні. Функція повертає ненульове значення, якщо копіювання вікна виконано успішно, інакше (наприклад, якщо нове зображення виходить за межі екрана) вікно не копіюється, а `movetext()` повертає нуль.

16.1.3. Виведення тексту у вікно екрана

Відображення символів, текстових рядків та числових даних в активному вікні програми можна виконувати трьома функціями консольного виведення, описи яких подано нижче. Для відображення даних використовуються кольори символів і фону, встановлені останніми викликами функцій `textcolor()`, `textbackground` або `textattr()`. Дані виводяться на екран, починаючи з поточної позиції курсора, яку можна змінювати за допомогою функції `gotoxy()`. Консольне виведення здійснюється в межах активного вікна. У разі досягнення правої межі вікна виведення тексту автоматично продовжується з початку наступного рядка. Коли текст виходить за нижню праву межу вікна, відбувається автоматичний вертикальний скролінг.

У заголовному файлі `<conio.h>` оголошено внутрішню цілочислову змінну `directvideo`, значення якої визначає спосіб реалізації виведення символів на екран. Якщо значення змінної `directvideo` нульове (таке значення встановлюється за замовчуванням), то операції виведення реалізуються через відповідні BIOS-переривання. Якщо ж `directvideo` надати значення `1`, то виведення даних буде виконуватись через безпосереднє звертання до відеопам'яті, що забезпечує вищу швидкодію, але вимагає, щоб відеосистема була IBM-сумісною.

Виведення окремого символа у поточну позицію активного текстового вікна здійснює функція

```
int  putch (int ch);
```

яка повертає код виведеного символа в разі успішного завершення та макроконстанту `EOF` за умови виникнення помилки.

Усі три функції консольного виведення дещо інакше реагують на керуючі символи (ескейп-послідовності), ніж відповідні функції високорівневого буферизованого виведення. Зокрема, виведення символа нового рядка `'\n'` (або `'\x0a'` у шістнадцятковому позначенні) переводить курсор на наступний рядок, але не встановлює його на початок цього рядка. Щоб перевести курсор на початок нового рядка, треба додатково вивести керуючий символ `'\r'` (або `'\x0d'`),

який встановлює курсор на початок поточного рядка. Виведення символа горизонтальної табуляції `'\t'` або вертикальної табуляції `'\v'` не переміщує курсор у відповідну табуляційну позицію, замість цього виводиться екранний символ, яким позначається код виведеної ескейп-послідовності.

Проілюструємо застосування `putch()` прикладом функції, яка здійснює різнокольорове виведення заданого символьного рядка. Кожен символ рядка функція відображає іншим кольором, а колір фону встановлює інверсним до кольору символа.

```
/*  Функція різнокольорового виведення тексту  */
void  OutColorText (char* text)
{
    char *ps=text;
    int col=0;                          /* колір символів */
  while (*ps!='\0') {
     textcolor(col);
     textbackground(~col);              /* інверсний колір фону */
     putch(*ps++);
     if (++col>15)                      /* зміна кольору */
       col=0;
  }
  putch('\r');                          /* перехід на початок */
  putch('\n');                          /* нового рядка */
}
```

Функція

```
int  cputs (const char* str);
```

виводить в активне вікно екрана символьний рядок, адресу початку якого задає вказівник `str`. Виведення починається з поточної позиції курсора. Функція повертає ASCII-код останнього виведеного на екран символа. На відміну від функції `puts()`, `cputs()` не переводить курсор на початок наступного рядка, а залишає його за останнім виведеним символом. Для встановлення курсора на початок нового екранного рядка необхідно, щоб символьний рядок закінчувався парою символів `"\n\r"` або `"\r\n"`, інакше одну з цих комбінацій потрібно виводити додатково.

Функція форматного виведення даних

```
int  cprintf (const char* format, ...)
```

є аналогом стандартної бібліотечної функції `printf()`, але додатково дає змогу використовувати можливості консольного виведення. Значення виразів зі списку виведення перетворюються відповідно до специфікацій форматного рядка `format` і виводяться в активному вікні. Як і в попередній функції, в `cprintf()` треба використовувати комбінацію символів `"\n\r"` або `"\r\n"` для переведення курсора на початок нового рядка. Функція `cprintf()` повертає загальну кількість виведених символів. Приклад звертання до функцій `cputs()` та `cprintf()` наведено далі.

16.1.4. Редагування рядків вікна

Три наступні функції працюють з рядками активного текстового вікна. Функція

```
void clreol (void);
```

витирає вміст частини рядка, починаючи від поточної позиції курсора до правої межі текстового вікна. Курсор зберігає свою позицію.

Друга функція витирання рядка:

```
void delline (void);
```

повністю видаляє рядок активного вікна, на якому в даний момент розміщений текстовий курсор. Всі наступні рядки цього вікна зсуваються на один угору. Курсор встановлюється на початок рядка, який пересунувся на місце видаленого.

Третя функція:

```
void insline (void)
```

вставляє порожній рядок в активне текстове вікно, зсуваючи всі наступні рядки вікна на один униз. Нижній рядок тексту, який внаслідок зсунення виходить за межі вікна, втрачається. Місце вставлення порожнього рядка визначає поточна позиція курсора, який переноситься на початок вставленого рядка.

Дію описаних функцій демонструють результати виконання наступної програми. У процесі роботи програма спочатку заповнює текстом невелике вікно, розташоване в центрі екрана. Після натискання користувачем на довільну клавішу відбувається витирання серединного рядка вікна, а перший рядок звільняється для нового запису. Перед завершенням роботи програма відновлює та очищає стандартне повноекранне вікно.

```
/*************************************/
/*  Редагування тексту у вікні екрана  */
/*************************************/
# include <conio.h>
void  ClearScreen(void);
void  main (void)
{
     int ww = 40, wh = 4;                                  /* розміри вікна */
     int x1, y1, x2, y2, r;
   x1 = (80-ww)/2+1;   x2 = x1+ww-1;     /* горизонтальні координати вікна */
   y1 = (25-wh)/2+1;   y2 = y1+wh-1;       /* вертикальні координати вікна */
   textcolor (14);  textbackground(1);
   window (x1,y1,x2,y2);  clrscr();                /* активізація вікна */
   for (r = 1;  r < wh;  r++)
      cprintf(" Рядок номер %d\n\r", r);
   cputs(" Останній рядок ");
   getch();                                          /* затримка зображення */
   gotoxy(1, wh/2);   delline();          /* витирання серединного рядка */
   gotoxy(1, 1);   insline();    /* вставлення порожнього першого рядка */
```

```
  cputs(" Новий рядок ");
  getch();
  ClearScreen();                    /* очищення повноекранного вікна */
}
void ClearScreen (void)
{
   textmode(LASTMODE);   clrscr();
}
```

Результати виконання: 1 – початкове заповнення текстового вікна; 2 – вміст вікна після виклику `delline();` 3 – вміст вікна після виклику `insline();` 4 – вміст вікна після редагування.

1)
```
Рядок номер 1
Рядок номер 2
Рядок номер 3
Останній рядок
```

2)
```
Рядок номер 1
Рядок номер 3
Останній рядок
```
порожній рядок

3) *порожній рядок*
```
Рядок номер 1
Рядок номер 3
Останній рядок
```

4)
```
Новий рядок
Рядок номер 2
Рядок номер 3
Останній рядок
```

16.1.5. Керування позицією та формою текстового курсора

Щоб встановити текстовий курсор у задану позицію активного вікна, треба звернутись до функції

```
void  gotoxy (int col, int row);
```

яка переміщує курсор у позицію `col` рядка `row` активного вікна екрана. Функція використовує віконні координати: координата (1, 1) відповідає лівому верхньому знакомісцю вікна. Якщо значення координат виходять за межі вікна, то курсор не переміщується.

Поточні координати курсора в активному вікні повертають дві функції:

```
int  wherex (void);
```

повертає горизонтальну координату (позицію курсора в рядку);

```
int  wherey (void);
```

повертає вертикальну координату курсора (номер рядка у вікні).

Для зміни форми та видимості текстового курсора призначена функція

```
void  _setcursortype (int cform);
```

параметр `cform` якої може приймати одне з трьох значень:

- `0` або `_NOCURSOR` – невидимий курсор;
- `1` або `_SOLIDCURSOR` – високий курсор (повне знакомісце);
- `2` або `_NORMALCURSOR` – звичайний курсор.

Два наведені нижче макроси призначені для керування видимістю текстового курсора: `CursorOff()` робить курсор невидимим, а `CursorOn()` відновлює звичайне зображення текстового курсора. Обидва макроси реалізують звертання до функції `_setcursortype()` із відповідним значенням параметра форми курсора.

```
/*  Макрос, що робить текстовий курсор невидимим  */
#define  CursorOff()   _setcursortype(_NOCURSOR)
/*  Макрос, що відновлює стандартне зображення курсора  */
#define  CursorOn()    _setcursortype(_NORMALCURSOR)
```

16.2. Консольне введення даних

16.2.1. Короткий опис процесу введення даних з клавіатури

За призначенням клавіші клавіатури персонального комп'ютера можна поділити на декілька груп:

1) символьні клавіші;
2) клавіші керування: *Home*, *End*, *PageUp*, *PageDn*, *Insert*, *Delete* та чотири клавіші стрілок;
3) функціональні клавіші: *F1*, *F2*, . . . *F12*;
4) шифт-клавіші: *RightShift*, *LeftShift*, *Alt*, *Ctrl*;
5) перемикальні (тригерні) клавіші: *NumLock*, *ScrollLock*, *CapsLock*, *Insert*;
6) спеціальні клавіші: *PrnScr*, *Pause/Break*.

Кожна клавіша має свій порядковий номер, який називають *скен-кодом* клавіші.

Керування роботою клавіатури здійснює спеціальний вбудований мікропроцесор – контролер клавіатури. У разі натискання на одну з клавіш або дозволену комбінацію клавіш, контролер клавіатури передає в центральний процесор запит на переривання і номер клавіші (скен-код). Скен-коди клавіш різних груп опрацьовуються по-різному.

Клавіші перших трьох груп називають клавішами з буферизацією розширеного коду. Коли натискають одну з цих клавіш, у спеціальну ділянку оперативної пам'яті – буфер клавіатури – заноситься двобайтовий код, що називається *BIOS-кодом* клавіші. У разі символьних клавіш молодший байт цього коду дорівнює ASCII-коду відповідного символа (крім натисненої клавіші враховується стан шифт-клавіш і перемикальних клавіш, щоб розрізнити великі та маленькі літери, а також спеціальні *Ctrl*-комбінації). У старший байт BIOS-коду символьних клавіш записується скен-код натисненої клавіші. Якщо ж натиснено функціональну клавішу, клавішу керування або їх комбінацію з клавішами *Alt*, *Ctrl* чи *Shift*, а також якщо натиснено символьну клавішу разом з клавішею *Alt*, то в буфер клавіатури записується BIOS-код, молодший байт якого дорівнює нулю, а старший – т. зв. *розширеному ASCII-коду* клавіші (табл. 16.3).

Буфер клавіатури розташований в оперативній пам'яті за адресою 0000:041Eh, він має ємність 32 байти і організований як циклічний список-черга. У буфері

Таблиця 16.3

Розширені ASCII-коди клавіш керування, функціональних клавіш і клавішних комбінацій

Клавіша або група клавіш	Окремо	Разом з клавішею		
		Shift	*Alt*	*Ctrl*
Insert	82	82	162	146
Delete	83	83	163	147
Home	71	71	151	119
End	79	79	159	17
PageUp	73	73	153	132
PageDown	81	81	161	118
Стрілка вліво	75	75	155	115
Стрілка вгору	72	72	152	141
Стрілка вниз	80	80	160	145
Стрілка вправо	77	77	157	116
F1 . . F10	59..68	84..93	104..113	94..103
F11, F12	133, 134	135, 136	139, 140	137, 138
верхній ряд символьних клавіш: *1 . . +*	–	–	120..131	–
другий ряд символьних клавіш: *Q . . }*	–	–	16..27	–
третій ряд символьних клавіш: *A . . "*	–	–	30..40	–
нижній ряд символьних клавіш: *Z . . ?*	–	–	44..53	–
клавіша ~	–	–	41	–
клавіша \	–	–	43	–
Tab	–	15	–	–

клавіатури може одночасно зберігатись до 15-ти символів (два додаткові байти виділено для поверненого символа). Запис символів у буфер виконує переривання 9h, а зчитування – функції переривання 16h. У випадку заповнення всього буфера без зчитування інформації подальший запис символів припиняється і виводиться звуковий сигнал.

Якщо окремо натиснути на одну із шифт-клавіш або перемикальних клавіш (їх називають клавішами-модифікаторами), то вміст буфера клавіатури не змінюється, оскільки ці клавіші (крім клавіші *Insert*) не використовуються самостійно, а призначені для розширення функцій інших клавіш.

Натискання на спеціальні клавіші та встановлені клавішні комбінації викликає відповідну реакцію операційної системи, зокрема:

- у разі натискання на клавішу *PrnScr* викликається BIOS-переривання 5h;
- комбінація *Ctrl+Alt+Del* запускає програму перезавантаження операційної системи;
- комбінація *Ctrl+Break* (а також *Ctrl+C*) встановлює спеціальний прапорець, що сигналізує про бажання користувача припинити виконання програми. Стан цього прапорця аналізують ряд бібліотечних функцій введення/виведення даних.

16.2.2. Функції консольного введення даних

На відміну від стандартних функцій потокоорієнтованого введення даних, функції введення, оголошені в `<conio.h>`, не виконують проміжної буферизації даних, а зчитують вхідну інформацію безпосередньо з буфера клавіатури. Ці функції забезпечують високу швидкодію процесів введення даних з клавіатури та надають користувачеві додаткові можливості для керування ходом виконання програми, оскільки вони реагують не тільки на символьні, а й на функціональні клавіші та клавіші керування (клавіші розширеної клавіатури). Перелік функцій консольного введення подано в табл. 16.4.

Таблиця 16.4

Функції консольного введення даних

Функція	Призначення
`cgets()`	Введення символьного рядка
`cscanf()`	Форматне введення даних
`getch()`	Введення одного символа без відображення
`getche()`	Введення одного символа з відображенням
`getpass()`	Введення пароля
`kbhit()`	Перевірка натискання довільної клавіші
`ungetch()`	Повернення символа у буфер клавіатури

Функції `getch()` та `ungetch()`. Найпопулярнішою функцією консольного введення є функція

```
int  getch (void);
```

яка призначена для зчитування коду символа натисненої клавіші без відображення цього символа на екрані. Код символа зчитується з буфера клавіатури, якщо ж буфер порожній, то виконання програми призупиняється і вона переходить у стан очікування натиснення клавіші. Незалежно від зчитаного символа позиція текстового курсора в активному вікні не змінюється. Функція повертає ASCII-код введеного символа, якщо натиснено символьну клавішу, або нуль, якщо натиснено одну з клавіш керування, функціональних клавіш або клавішних комбінацій з набору розширеної клавіатури (див. табл. 16.3). У цьому випадку потрібно ще раз викликати `getch()`, щоб отримати розширений ASCII-код клавіші.

Проілюструємо застосування `getch()` на прикладі функції `MoveCursor()`, яка відстежує натискання на клавішу *Esc*, клавіші стрілок і клавіші *Home* та *End* (всі інші клавіші ігноруються). У разі натискання на клавіші стрілок функція пересуває текстовий курсор у відповідну позицію активного вікна (якщо досягнуто межі вікна, то рух курсора блокується). Клавішею *Home* курсор встановлюється у ліве верхнє знакомісце, а клавішею *End* – в останню позицію вікна. У разі натискання на *Esc* функція повертає `0`, якщо натиснено клавішу переміщення курсора, то функція повертає `1`, якщо ж натиснено іншу клавішу, то `MoveCursor()` повертає `-1`.

```
/*  Функція переміщення текстового курсора  */
#include <conio.h>
#define Esc 0x1b                              /* ASCII-код клавіші Esc */
int  MoveCursor (void)
{
     int key, x, y;
     int winw, winh;                          /* розміри вікна */
     struct text_info win;                    /* структура параметрів вікна */
  key = getch();                                /* зчитування коду клавіші */
  if (key == Esc)  return 0;                              /* натиснено Esc */
  else  if (key != 0)  return -1;          /* натиснено символьну клавішу */
  gettextinfo(&win);             /* визначення параметрів активного вікна */
  winw = win.winright - win.winleft+1;                    /* ширина вікна */
  winh = win.winbottom - win.wintop+1;                    /* висота вікна */
  x = win.curx;  y=win.cury;                /* поточні координати курсора */
  switch (key = getch()) {          /* зчитування розширеного коду клавіші */
    case 77: if (x < winw) x++;  break;                   /* натиснено → */
    case 80: if (y < winh) y++;  break;                   /* натиснено ↓ */
    case 75: if (x > 1) x--;  break;                      /* натиснено ← */
    case 72: if (y > 1) y--;  break;                      /* натиснено ↑ */
    case 71: x = 1;  y = 1;  break;              /* натиснено клавішу Home */
    case 79: x = winw;  y = winh;  break;                 /* натиснено End */
    default: return -1;              /* натиснено іншу клавішу керування */
  }
  gotoxy(x,y);                      /* перенесення курсора в нову позицію */
  return 1;
}
```

Функція

```
int  ungetch (int symb);
```

заносить (повертає) заданий символ `symb` безпосередньо у буфер клавіатури. Наступна операція зчитування даних введе цей символ першим. Повертати можна тільки один символ, а повторно звертатись до `ungetch()` тільки тоді, коли попередній повернений символ зчитано з буфера клавіатури. За умови успішного виконання значенням функції `ungetch()` є код поверненого символа, якщо ж зафіксовано помилку, то – `EOF`.

Функція `kbhit()`. Ця функція перевіряє, чи відбувалось натискання на клавіші, точніше, чи є в буфері клавіатури незчитані символи. Прототип її такий:

```
int  kbhit (void);
```

Функція повертає нуль, якщо буфер клавіатури порожній, та ненульове значення, якщо в буфері є якісь символи. Від усіх інших функцій звертання до клавіатури `kbhit()` відрізняється тим, що не зупиняє виконання програми, не очікує натискання на клавіші та не зчитує символів з буфера клавіатури.

Використовуючи функцію kbhit(), можна забезпечити циклічне виконання потрібних дій, поки не буде натиснено довільну або задану клавішу:

```
/*  Приклад циклу, що завершується, коли натиснено якусь клавішу  */

do {
    . . .                    /* дії, що виконуються циклічно */
} while (!kbhit());
```

У записаній нижче функції ClearKeyBuffer() функцію kbhit() використано для очищення (скидання) буфера клавіатури.

```
/*  Функція скидання буфера клавіатури  */

void ClearKeyBuffer (void)
{
   while (kbhit())           /* поки в буфері клавіатури є символи */
     getch();                /* зчитуємо їх */
}
```

Функції введення даних з відображенням на екрані. Наступні три функції: getche(), cgets() та cscanf() здійснюють введення даних з клавіатури та відтворюють на екрані введені символи. Відображення зчитаних символів здійснюється в межах активного вікна, починаючи від поточної позиції курсора. Використовуються атрибути (кольори), встановлені для даного вікна останніми.

Функція

```
int  getche (void);
```

є аналогом описаної вище функції getch(), але додатково реалізує відображення введеного символа (т. зв. *ехо-друк*). Якщо натиснено клавішу *Enter*, то вводиться символ "повернення каретки" '\r', тому курсор переводиться на початок даного (а не наступного) рядка. У випадку натискання на клавішу керування або функціональну клавішу на екран виводиться символ пробілу (так відображається зчитаний символ з кодом 0), а наступний виклик функції getche() відображає на екрані символ, код якого дорівнює розширеному ASCII-коду натисненої клавіші. Текстовий курсор після введення встановлюється за відображеним символом.

Консольне введення символьних рядків виконує функція

```
char*  cgets (char* buf);
```

яка записує введений з клавіатури рядок символів у ділянку оперативної пам'яті, адресу початку якої задає вказівник buf. Перші два байти буфера buf використовуються для збереження параметрів рядка. У buf[0] перед викликом функції треба занести значення, яке обмежуватиме довжину введеного рядка (з урахуванням байта для '\0'). У buf[1] після завершення введення буде записано реальну довжину введеного рядка (без '\0'). Символи рядка заносяться в буфер, починаючи з елемента buf[2]. Можна ввести не більше, ніж buf[0]-1 символів, у кінець введеного рядка автоматично

записується '\0'. Процес введення рядка завершує натискання на клавішу *Enter*. Текстовий курсор залишається за останнім відтвореним на екрані символом. За умови успішного виконання функція повертає вказівник на перший символ введеного рядка, тобто адресу buf+2, а в разі помилки – NULL.

Функція cgets() не тільки відображає введений рядок у межах активного вікна, використовуючи поточні кольори, але й має ряд додаткових можливостей порівняно з функцією високорівневого потокоорієнтованого введення gets():

- можна вводити з клавіатури рядки, що містять до 254-х символів (буферизоване введення обмежує довжину рядка 127-ма символами);
- задання максимальної довжини рядка в buf[0] гарантує захист від введення надлишкових символів;
- відразу після введення відомою є довжина рядка;
- курсор залишається за останнім символом введеного рядка.

Наведена далі функція InputString() заносить у задану ділянку пам'яті введений з клавіатури рядок символів, відображаючи його в активному вікні. Параметр size обмежує розмір рядка введення, а вказівник len задає адресу, за якою записується реальна довжина введеного рядка.

```
/*  Функція консольного введення символьного рядка  */
char * InputString (char * stbuf, int maxlen, int * len)
{
  *stbuf=maxlen+1;                    /* обмеження довжини */
  cgets(stbuf);
  *len=*(stbuf+1);                    /* довжина введеного рядка */
  return stbuf+2;                     /* початок введеного рядка */
}
```

Функція

```
int  cscanf (const char* format, ... );
```

виконує форматне введення даних з відображенням зчитаних з клавіатури символів у активному вікні. За форматними перетвореннями даних cscanf() є повним аналогом функції високорівневого введення scanf(). Водночас cscanf() не здійснює внутрішньої буферизації і опрацьовує кожен символ відразу після натискання на його клавішу. Введення завершується, коли зчитано поле даних, що відповідає останній специфікації формату, або коли введено некоректний для заданої специфікації символ. Після останнього поля даних можна ввести довільний символ-роздільник (не обов'язково натискати на *Enter*). Цей символ відображається на екрані, а його код повертається у буфер клавіатури. Якщо в процесі введення окремі поля даних відокремлюються натисканням на клавішу *Enter* (або цією клавішею завершується введення даних), то текстовий курсор переводиться на початок поточного (не наступного) рядка, інакше курсор встановлюється за останнім відтвореним символом. Функція повертає кількість успішно введених і записаних у пам'ять даних.

16.2.3. Аналіз стану клавіш-модифікаторів

Використання шифт-клавіш: *RightShift*, *LeftShift*, *Alt*, *Ctrl* і перемикальних клавіш: *NumLock*, *ScrollLock*, *CapsLock*, *Insert* – їх спільно називають клавішами-модифікаторами – розширює функціональні можливості основних клавіш.

Хоча клавіша *Insert* належить до клавіш з розширеними ASCII-кодами, в багатьох випадках її використовують як тригерну клавішу, фіксуючи включення або виключення режиму вставляння.

Інформацію про поточний стан шифт- та перемикальних клавіш, а також клавіші *Insert* можна отримати з області даних BIOS, зчитуючи значення двох послідовних байтів за адресами 0000:0417h та 0000:0418h (табл. 16.5). Окремі біти цих байтів змінюються, коли відбувається натискання або відпускання відповідних клавіш. Для індикації стану кожної з *Shift*-клавіш відводиться один біт, який встановлюється в 1, коли клавішу натиснено, і скидається в 0, коли клавішу відпущено. За кожною з тригерних клавіш: *NumLock*, *ScrollLock*, *CapsLock* та *Insert* закріплено по два біти, що мають однакові номери в обох байтах слова стану клавіш. Значення бітів байта, розташованого за адресою 0:0418h змінюються на протилежні при кожному натисканні та кожному відпусканні відповідної клавіші (фіксується стан "Натиснено" або "Відпущено"). Значення бітів молодшого байта слова стану клавіш змінюються у момент натискання клавіші (фіксується стан "Включено" або "Виключено").

Таблиця 16.5

Структура BIOS-слова стану клавіш-модифікаторів

Байт за адресою 0000:0417h		Байт за адресою 0000:0418h	
Номер біта	Призначення	Номер біта	Призначення
0	Натиснено праву клавішу *Shift*	0	Натиснено ліву клавішу *Ctrl*
1	Натиснено ліву клавішу *Shift*	1	Натиснено ліву клавішу *Alt*
2	Натиснено одну з клавіш *Ctrl*	2	Натиснено клавішу *SysReg*
3	Натиснено одну з клавіш *Alt*	3	Включено клавішу *Pause*
4	Включено клавішу *ScrollLock*	4	Натиснено клавішу *ScrollLock*
5	Включено клавішу *NumLock*	5	Натиснено клавішу *NumLock*
6	Включено клавішу *CapsLock*	6	Натиснено клавішу *CapsLock*
7	Включено клавішу *Insert*	7	Натиснено клавішу *Insert*

Проілюструємо звертання до слова стану клавіш-модифікаторів прикладом функції `RightCtrlDown()`, яка перевіряє, чи натиснено праву клавішу *Ctrl*. Індикатором натискання клавіші слугує 1 у другому біті байта, що має адресу 0:0417h, за умови, що наймолодший біт наступного байта дорівнює 0 (тобто не натиснено ліву клавішу *Ctrl*). Функція повертає значення 1, якщо в момент перевірки натиснено праву клавішу *Ctrl*, і значення 0, якщо клавішу не натиснено.

```
/*  Функція перевірки натискання правої клавіші Ctrl  */
#include <dos.h>
int  RightCtrlDown (void)
{
  unsigned char far * kstat_byte1 =          /* адреса молодшого байта */
          (unsigned char far *)MK_FP(0x0, 0x0417);
  unsigned char far * kstat_byte2 = kstat_byte1+1;     /* адреса старшого
                                                  байта слова стану клавіш */
    return  (kstat_byte1 & 0x4) && !(kstat_byte2 & 0x1);
}
```

Аналіз стану шифт- та перемикальних клавіш розширює можливості щодо керування роботою програми, зокрема дає змогу створювати власні комбінації клавіш певного призначення.

Запитання та завдання для самоконтролю

1. Яке призначення функцій, оголошених у заголовному файлі `<conio.h>`? Чи належать вони до стандартизованих функцій мови С?
2. Які текстові режими підтримують відеоадаптери VGA/SVGA? Як змінити текстовий режим на інший?
3. Які функції дають змогу керувати кольором символів, що виводяться на екран? Наведіть приклад звертання до цих функцій, у результаті якого виведення тексту буде виконуватись зеленим кольором на білому фоні.
4. Що називають текстовим вікном? Яка функція відкриває вікно? Що відбувається в результаті відкриття текстового вікна?
5. Як намалювати рамку вікна?
6. Які характеристики поточного текстового режиму можна отримати через функцію `gettextinfo()`? Що є параметром даної функції?
7. Як зберегти, а потім відновити текстове вікно? Чи стає активним відновлене вікно?
8. Що виконує функція `putch()`? Чим вона відрізняється від функції `putchar()`?
9. Які функції призначені для виведення текстових повідомлень в активному вікні екрана? У чому їх особливості?
10. За допомогою яких функцій можна керувати позицією і формою текстового курсора?
11. На які групи поділяються клавіші клавіатури персонального комп'ютера? Яке призначення кожної з клавішних груп?
12. Що відбувається у разі натискання на одну із символьних клавіш? З чого складається BIOS-код символьних клавіш?
13. Що записується у буфер клавіатури в разі натискання на одну з клавіш керування чи функціональних клавіш? З чого складається BIOS-код цих клавіш?
14. Як отримати код натисненої клавіші керування або функціональної клавіші? Напишіть фрагмент програми, який реалізує перевірку, чи натиснено функціональну клавішу *F1*?

15. Нижче подано два варіанти оператора циклу, в якому виконуються певні задані дії і аналізується код натисненої клавіші. Яку принципову відмінність у реалізацію дій тіла циклу внесло звертання до функції `kbhit()`?

1)
```
do {
   . . .   /* задані дії */
  key = getch();
  . . .   /* дії, що залежать
    від натисненої клавіші */
} while (key!=Esc);
```

2)
```
do {
   . . .   /* задані дії */
  if (kbhit()) {
    key = getch();
    . . .   /* певні дії */
  }
} while (key!=Esc);
```

16. У програмі оголошено масив і вказівник:

```
char  inst[200], *ps=inst;
```

та виконано присвоєння і звертання до функції:

```
*ps=50;
ps=cgets(inst);
```

Як заповниться масив `inst`, якщо з клавіатури буде введено слово `Кінець`? Яким буде значення `inst[1]`? Чи зміниться значення вказівника `ps`?

17. Для введення значень двох цілочислових змінних `x1` та `x2` у програмі використано функцію

```
cscanf("%d%d", &x1, &x2);
```

Чи можна значення цих змінних ввести так:

```
28_50_
```

(де знаком _ позначено символ пробілу), тобто без натискання на *Enter*?
Як будуть відображені на екрані введені дані, якщо після кожного з них натискати клавішу *Enter*?

18. Які клавіші називають тригерними? Як можна дізнатись, чи включено клавішу *Insert*?

Запрограмуйте наведені нижче задачі, використовуючи функції консольного введення та відображення текстових даних

19. У правій нижній частині екрана сформувати текстове вікно, оточене рамкою. Вивести у вікні коротку інформацію про зміст програми та її автора. Рядки тексту повинні бути відцентровані у вікні по горизонталі та по вертикалі.

20. З клавіатури ввести послідовність символьних рядків, відобразивши їх різними кольорами у відкритому текстовому вікні. Організувати редагування введеного тексту, використовуючи клавіші *Delete* та *Insert*: при натисканні на *Delete* поточний рядок вікна витирається, а при натисканні на *Insert* у позиції розміщення курсора вставляється вільний рядок для введення. Переміщенням курсора керувати за допомогою клавіш стрілок.

Розділ 17

ДИРЕКТИВИ ПРЕПРОЦЕСОРА

У цьому розділі:

- Призначення директив препроцесора
- Директива приєднання текстових файлів `#include`
- Директиви макрозамін `#define` та `#undef`: текстові макропідстановки, особливості макросів з параметрами, використання операцій `#` та `##` для розширення можливостей макропідстановок
- Організація умовної компіляції – директиви: `#if`, `#else`, `#elif`, `#endif` та `#ifdef`/`#ifndef`, операція `defined`
- Вбудовані макроси: `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, `__STDC__`
- Перегляд результатів препроцесорного опрацювання програми
- Допоміжні директиви: `#line`, `#error`, `#pragma`

Окрім відомих вже директив препроцесора `#include` та `#define`, що використовуються практично в усіх програмах, мова С підтримує ряд інших директив, які теж призначені для керування процесом *препроцесування*, тобто попереднього опрацювання тексту програми.

Препроцесор – це спеціальна програма, що опрацьовує текстовий файл С-програми перед компіляцією. Він входить як обов'язковий компонент до складу компілятора і/або середовища програмування мови С. Результатом роботи препроцесора є готовий до компіляції текст програми, в якому реалізовані всі задані директиви (інструкції) попереднього опрацювання програми.

Найчастіше на етапі препроцесування виконуються такі дії:

1) включення до складу програми текстів зі заданих файлів;
2) вилучення з тексту програми тих частин, які не повинні компілюватись у даній реалізації (умовна компіляція);
3) заміна в програмі макроімен на задані вирази (тексти) підстановки;
4) реалізація макропідстановок (макросів) з параметрами.

Стандарт мови С підтримує такі директиви препроцесора:

`#define`	`#elif`	`#else`	`#endif`	`#error`	`#if`
`#ifdef`	`#ifndef`	`#include`	`#line`	`#pragma`	`#undef`

Перш ніж розглянути правила та особливості застосування перелічених директив, нагадаємо, що кожна директива препроцесора записується в окремому рядку, першим символом якого є знак `#`. Після директив знак `;` не ставиться.

17.1. Директива включення `#include`

Директива `#include` призначена для приєднання до програми текстового файла, ім'я якого задається в цій директиві (такий файл називають *файлом включення*). Текст з файла вставляється в програму в тому місці, де була записана директива `#include`. Директива допускає дві форми запису імен файлів:

```
#include  <ім'я_файла>          /* ім'я в кутових дужках */
#include  "ім'я_файла"          /* ім'я в лапках */
```

Форма запису імені файла визначає, де відбуватиметься його пошук. Якщо ім'я файла записане в кутових дужках, то такий файл повинен зберігатись у спеціальному каталозі середовища програмування (для Borland C це каталог INCLUDE) або в іншому визначеному для компілятора каталозі. Якщо ж ім'я файла задається в лапках, то більшість компіляторів розпочинають пошук такого файла з поточного активного каталога. Якщо файл там не знайдено, то пошук продовжується у каталозі, що відповідає варіанту кутових дужок. Здебільшого в кутових дужках вказують імена стандартних заголовних файлів бібліотечних функцій, а в лапках – імена користувацьких файлів, які повинні бути приєднані до даної програми. Проте це не є обов'язковим правилом.

Практика формування власних файлів включення широко застосовується в програмуванні мовою С, насамперед у разі створення багатофайлових програм. Найчастіше в заголовні файли `*.h` виносять прототипи функцій, шаблони структур, оголошення користувацьких типів, визначення макросів тощо. Сформовані заголовні файли підключають до кожного програмного файла, що дає змогу автономно компілювати кожен з програмних файлів. Доцільно також створювати файли включення, в які записувати власні набори функцій, об'єднані за певним призначенням (наприклад, набір власних функцій для введення даних з клавіатури чи набір функцій, призначених для керування "мишкою"). Тоді однією директивою `#include` ці функції можна буде підключити до кожної програми, де вони використовуються.

Файли, що приєднуються директивою `#include`, можуть містити в своєму тексті інші директиви `#include`. Згідно зі стандартом С-89 компілятори повинні підтримувати не менше, ніж 8 рівнів вкладення, а згідно з новим стандартом – понад 15 рівнів.

17.2. Директиви макропідстановок #define та #undef

Директива препроцесора `#define` виконує задані заміни (*макропідстановки*) в тексті програми, їх називають також *макророзширеннями* або *макрозамінами*. Основним призначенням `#define` є створення іменованих констант (макроконстант) і функціональних макровизначень (макросів) з параметрами.

17.2.1. Текстові макропідстановки

Для виконання заданих макрозамін у тексті програми використовують директиву `#define`, що має таку форму:

`#define` *ім'я_макроса текст_заміни*

Ім'я макроса записується за правилами запису ідентифікаторів і задає лексему програми, яка має бути замінена вказаним далі *текстом_заміни*. Між іменем макроса та текстом макропідстановки може стояти довільна кількість символів пробілу. Кінець тексту заміни задає символ кінця рядка.

Хоча ім'я макроса може бути довільним ідентифікатором, програмісти здебільшого застосовують в іменах макросів заголовні літери, щоб у тексті програми вирізнити їх з-поміж звичайних імен.

У наших програмах неодноразово використовувалась директива `#define` для створення іменованих констант. Наведемо ще один приклад. Нехай оголошення масивів у програмі зроблено з використанням макроконстанти `MAXSIZE`:

```
#define  MAXSIZE  200
   . . .
  int  inarr[MAXSIZE],
       newarr[2*MAXSIZE+50];
  double  matr[MAXSIZE/4][MAXSIZE/2];
```

Після препроцесорного опрацювання програми ці оголошення будуть такими:

```
  int  inarr[200],
       newarr[2*200+50];
  double  matr[200/4][200/2];
```

У разі потреби достатньо лише змінити значення константи в директиві `#define`, щоб відповідно змінились розміри всіх масивів даного оголошення.

У текстах макропідстановок можна використовувати імена макросів, визначених раніше, наприклад:

```
#define  MIN  15
#define  MAX  20*MIN
#define  RANGE  (MAX-MIN)
```

Ще раз підкреслимо, що `#define` виконує просту заміну заданого ідентифікатора вказаним текстом. Тому в тексті підстановки можна записати частину оператора чи навіть групу операторів. Припустимо, що в програмі часто доводиться висвітлювати повідомлення про помилку у введених даних. Тоді доцільно скористатись таким макросом:

```
#define  ERROR_MSG  printf("Неправильні вхідні дані! \n")
```

Після препроцесування у всьому тексті програми оператор:

```
ERROR_MSG;
```

буде замінено викликом функції:

```
printf("Неправильні вхідні дані! \n");
```

У разі довгих текстів заміни одного рядка для запису `#define` може бути замало. Дозволено продовжувати текст заміни в наступному рядку – для цього попередній рядок треба закінчити символом лівої косої риски. Наведемо приклад:

```
#define  ERR_MSG_LONG  printf("\t\t Неправильні вхідні дані! \
 \n Повторіть, будь-ласка, введення: ")
```

Треба враховувати, що всі символи пробілу, записані перед косою рискою і на початку рядка продовження, будуть включені в текст підстановки.

17.2.2. Макроси з параметрами

Макроси з параметрами, які називають також *функціональними макровизначеннями* або просто *макросами*, ми вже розглядали в розділі 4. Нагадаємо їх основні властивості.

Макроси з параметрами дають змогу здійснювати макропідстановки, подібні до викликів функцій. Формальні параметри, вказані в списку `#define`, під час препроцесування замінюються фактичними виразами, заданими в звертанні до макроса.

Синтаксис макроса з параметрами такий:

`#define` *ім'я_макроса*(*список параметрів*) *вираз_для_заміни*

Між іменем макроса і дужкою, за якою записується список параметрів, не повинно бути символів пробілу, інакше (*список параметрів*) буде сприйнятий як початок тексту макрозаміни.

Наведемо приклад макроса, що виконує піднесення до куба заданого аргументу:

```
#define  CUBE(x)  ((x)*(x)*(x))        /* макрос, що реалізує x³ */
```

Цей макрос можна застосовувати для обчислення значення куба довільного арифметичного виразу. Наприклад:

```
/*  Використання макроса CUBE(x)  */

     long numb = 50;
     double z = 0.863;
```

```
printf (" Куб numb => %ld \n Куб z => %1.5lf \n Куб z*numb =>"
        " %1.4le \n", CUBE(numb), CUBE(z), CUBE(z*numb));
```

На екран буде виведено:

```
Куб numb => 125000
Куб z => 0.64274
Куб z*numb => 8.0342e+04
```

У виразі підстановки макроса кожен з параметрів і весь вираз слід брати в дужки, щоб уникнути можливих помилок у випадках, коли аргументи виклику є виразами або, коли сам макрос є елементом іншого виразу.

Якби вираз підстановки `CUBE(x)` був записаний так: `(x*x*x)`, то звертання `CUBE(z+1)` було б замінене помилковим виразом `(z+1*z+1*z+1)`. Якщо ж опустити зовнішні дужки, то помилковим буде значення виразу `1.0/CUBE(z)`, оскільки результат підстановки: `1.0/(z)*(z)*(z)` насправді відповідає виразу `(z)`.

Виконані макропідстановки можна побачити в файлі результатів препроцесування. Питання створення та перегляду такого файла розглянемо в параграфі 17.5. У наведеному там прикладі проілюстровано декілька реалізацій директиви `#define`.

Оскільки директива `#define` призначена для виконання текстових підстановок, то можна створити і такий макрос:

```
#define  FUN(f,arg)  f(arg)          /* макрос, що викликає функцію */
```

що дає змогу реалізувати звертання до різних функцій. Вираз:

```
FUN(sin,x/3) + FUN(log,x-0.54) + FUN(sqrt,2*x)
```

на етапі препроцесування буде замінено виразом:

```
sin(x/3) + log(x-0.54) + sqrt(2*x)
```

Як вже зазначалося, макроси з параметрами нагадують функції. Їх часто використовують саме для заміни функцій. Це дає дві істотні переваги. По-перше, макрос вбудовується в тіло програми на етапі препроцесування, отже, під час виконання програми не витрачається час на передавання параметрів і повернення значень, як це відбувається у разі виклику функцій. По-друге, аргументи, що вказуються у звертанні до макросів, можуть мати довільний тип. Тип аргументів макроса одночасно визначає і тип усього виразу макропідстановки. Зокрема вираз `CUBE(numb)` має тип `long`, а вираз `CUBE(z)` – тип `double`. Водночас типи параметрів функції і тип її значення задаються незмінними в оголошенні кожної функції.

Слід пам'ятати, що неправильний запис виразу підстановки та неконтрольованість параметрів макросів можуть призвести до помилкових результатів у макрозамінах, тому оголошення макросів і звертання до них потребують підвищеної уваги.

Переваги макросів і функцій суміщують у собі `inline` функції (див. параграф 11.4), які широко використовуються у мові C++ та запроваджені у С-програми новим стандартом С-99.

17.2.3. Операції # та ## у директиві #define

Операції # та ## розширюють можливості директиви #define. Їх можна використовувати тільки в тексті макропідстановки.

Операція # називається операцією *перетворення в стрінг* – вона перетворює в символьний рядок аргумент, перед яким записано знак #. Проілюструємо використання цієї операції таким прикладом:

```
/*  Використання операції # - перетворення в стрінг  */
#define  OUT(x)   printf("%s = %d \n", #x, x )
     . . .
   int  kr=25;
 OUT(kr);        OUT(kr/3);
 OUT(5*kr%10);
```

У процесі виконання цього фрагмента програми на екран буде виведено:

```
kr = 25
kr/3 = 8
5*kr%10 = 5
```

Перший параметр списку виведення функції printf(), яку записано в тексті макропідстановки макроса OUT(), є символьним рядком, сформованим за допомогою операції #x. Тому в звертанні OUT(kr) вираз #kr замінюється стрінгом "kr", в наступному звертанні OUT(kr/3) аргумент kr/3 замінюється операцією # на рядок "kr/3" і т. д.

Як засвідчує попередній приклад, за допомогою операції # можна легко ввести у рядки виведення імена змінних чи навіть цілі вирази. Зокрема, наше попереднє повідомлення про неправильне введення даних можна доповнити іменем змінної, щоб підвищити інформаційність повідомлення:

```
#define ERR_VAR_MSG(var) puts("Неправильне значення параметра "#var)
```

Тоді макрос:

```
ERR_VAR_MSG(speed);
```

буде замінено оператором:

```
puts("Неправильне значення параметра ""speed");
```

А всі послідовно записані символьні рядки (між ними можуть бути пробіли, символи нового рядка або інші символи-роздільники) компілятор об'єднує в один спільний рядок, тому остаточно заміна макроса буде такою:

```
puts("Неправильне значення параметра speed");
```

Операцію ## називають операцією *конкатенації* або *склеювання*. Вона призначена для об'єднання (конкатенації) двох лексем в одну нову лексему.

Наведемо приклад склеювання ідентифікаторів змінних:

```
/*  Використання операції конкатенації ##  */
#define  OUT3(x,y)   printf(" %d  %d  %d\n", x, y, x##y )
      . . .
   int  sm = 5,  ax = 6;
   int  smax = 100;
  OUT3(sm, ax);
```

Виконання даного фрагмента програми виведе на екран три значення :

```
5  6  100
```

Перші два числа є значеннями змінних `sm` та `ax`, а третє число – значення змінної `smax`. У процесі виконання замін у макросі `OUT3(sm,ax)` відбулось склеювання лексем: `sm##ax`, що створило новий ідентифікатор – `smax`.

У наступному прикладі записано макрос, який спочатку виводить ім'я, а потім – значення новоствореної лексеми.

```
/*  Спільне використання операцій # та ##  */
#define  OUTNEW(x,y)   printf ("\t %s => %d \n", #x#y, x##y )
      . . .
  OUTNEW(sm, ax);
```

Результат виконання:

```
    smax => 100
```

Під час підстановок у `printf()` вираз `#x#y` замінюється на `"sm""ax"`, з якого формується спільний рядок – `"smax"`, а вираз `x##y` буде замінено лексемою `smax`. Наприкінці заміна макроса `OUTNEW(sm, ax)` набуде вигляду оператора:

```
printf("\t %s => %d \n", "smax", smax );
```

17.2.4. Директива #undef

Дана директива вказує препроцесору про відміну макровизначення, попередньо встановленого через `#define`. Вона записується так:

`#undef` *ім'я_макроса*

У частині програми, записаній після `#undef`, вказаний макрос вважається невизначеним і жодні заміни не виконуються.

Ось приклад:

```
/*  Використання #undef для локалізації макроконстанти  */
#define LIMIT 500
    . . .     /* частина програми, де LIMIT замінюється значенням 500 */
#undef LIMIT
    . . .                  /* частина програми, де LIMIT невизначена */
#define LIMIT 850
    . . .           /* у цій частині LIMIT замінюється значенням 850 */
```

17.3. Директиви умовної компіляції

Для організації умовної компіляції використовують директиви препроцесора: `#if`, `#else`, `#elif`, `#endif`, а також директиви: `#ifdef`, `#ifndef` та операцію `defined`. Перелічені директиви призначені для вибіркової компіляції окремих частин програми. Вони дають змогу опустити компіляцію тих частин (фрагментів) програми, які не відповідають заданим умовам. Умовна компіляція є зручною для налагодження програм. Крім цього, її широко застосовують у комерційних програмних продуктах, в яких передбачають декілька спеціальних версій загальної програми.

17.3.1. Директиви `#if`, `#else`, `#elif` та `#endif`

Синтаксично директиви умовної компіляції, які вказують, чи буде компілюватись заданий програмний фрагмент, подібні до оператора `if`. Вони можуть встановлювати скорочену форму умовної компіляції:

```
#if константний_вираз
        фрагмент_програми
#endif
```

або повну:

```
#if константний_вираз
        фрагмент_програми_1
#else
        фрагмент_програми_2
#endif
```

Константний_вираз директиви `#if`, який задає умову компіляції, може складатись з констант або ідентифікаторів макропідстановок. У разі короткої форми умовної компіляції фрагмент програми між `#if` та `#endif` буде компілюватись тільки за умови, що значення константного виразу відмінне від нуля (істинне). У разі повної форми залежно від значення виразу `#if` буде компілюватись або *фрагмент_програми_1* (коли значення виразу ненульове) або *фрагмент_програми_2* (коли значення виразу дорівнює нулю). Кінець частини програми, яка підлягає умовній компіляції, задає директива `#endif`.

Наведемо приклад.

```
/*  Організація умовної компіляції через #if */
#define SIZE 300
    unsigned arr[SIZE];
   . . .              /* початкова частина програми */
#if  SIZE > 500
   printf ("Масив великої розмірності.\n");
     . . .            /* оператори, пов'язані тільки з великим масивом */
```

```
#else
   printf ("Масив малої розмірності.\n");
     . . .              /* оператори, пов'язані тільки з малим масивом */
#endif
```

Для встановленого через `#define` значення макроконстанти `SIZE (300)` частина програми між `#if` та `#else` компілюватись не буде. У виконавчий код програми потраплять тільки оператори, записані після `#else` та всі фрагменти програми, що не підлягали умовній компіляції.

Директива `#elif` є спрощеним варіантом конструкції `#else – #if`. Синтаксис її такий самий, як і директиви `#if`:

```
#elif константний_вираз
        фрагмент_програми
```

Якщо заданий *константний_вираз* не дорівнює нулю, то записаний далі *фрагмент_програми* передається на компіляцію, інакше він пропускається. За допомогою `#elif` можна легко створювати послідовності перевірок:

```
#if вираз_1
        фрагмент_програми_1
#elif вираз_2
        фрагмент_програми_2
    . . .
#elif вираз_k
        фрагмент_програми_k
#else
        альтернативний_фрагмент
#endif
```

Першим перевіряється константний *вираз_1*. Якщо він хибний (дорівнює нулю), то перевіряється *вираз_2* і т. д., доки не буде знайдено вираз із ненульовим значенням. Якщо такий варіант знайдено, то відповідний фрагмент програми передається на компіляцію, а перевірка умов завершується. Якщо ж всі вирази хибні, то компілюватись буде фрагмент програми, що відповідає варіанту `#else` (за умови, що такий альтернативний варіант є в програмі). Таким чином, на компіляцію буде передано тільки один (або не передано жодного) з усіх фрагментів програми, що підлягають умовній компіляції.

Розглянемо приклад. Нехай для певної програми розроблено набір заголовних файлів `ukrtext.h`, `engtext.h`, `rustext.h` та `germtext.h`, у яких записано текстові повідомлення, функції тощо, необхідні для реалізації інтерфейса програми відповідно українською, англійською чи іншою мовою. Залежно від того, яку версію виконавчого коду програми треба створити, вибирається один із цих файлів. Вибір відповідного заголовного файла виконується через макрос `LANGUAGE`.

```
/*  Вибір одного зі заданого набору заголовних файлів  */
#if  LANGUAGE == ENG
  #include  <engtext.h>
#elif  LANGUAGE== RUS
  #include  <rustext.h>
#elif  LANGUAGE== GERM
  #include  <germtext.h>
#else
  #include  <ukrtext.h>
#endif
```

Макроси: `ENG`, `RUS` та `GERM` повинні бути попередньо визначені директивою `#define`. Якщо параметр `LANGUAGE` перед `#if` не отримав значення якогось з цих макросів (варіант `#else`), то встановлюється україномовний інтерфейс. Для вибору іншої мови повідомлень, наприклад, англійської, достатньо вказати директиву:

```
#define  LANGUAGE  ENG
```

17.3.2. Директиви #ifdef, #ifndef

Ще одним способом організації умовної компіляції є використання директив `#ifdef` (*if defined* – за умови визначеності) та `#ifndef` (*if not defined* – за умови невизначеності). Синтаксис їх однаковий:

```
#ifdef ім'я_макроса
        фрагмент_програми
#endif
```

та

```
#ifndef ім'я_макроса
        фрагмент_програми
#endif
```

Обидві директиви перевіряють, чи макрос, ім'я якого задається в цій директиві, був попередньо визначений директивою `#define`. При цьому не вимагається, щоб макрос мав певне значення, достатньо простого оголошення:

```
#define ім'я_макроса
```

Фрагмент програми, обмежений директивами `#ifdef ... #endif`, буде компілюватись за умови, що *ім'я_макроса* попередньо визначене. Дія директиви `#ifndef` протилежна – *фрагмент_програми* передається на компіляцію, якщо заданий макрос не визначений.

Як вже зазначалось, умовну компіляцію часто використовують для налагодження програм. Зокрема, в текст програми можна включити ряд контрольних перевірок і по-

відомлень, що допоможуть відстежити процес виконання програми. Наприклад :

```
/*  Організація умовної компіляції через #ifdef */
#define  DEBUG
     . . .                    /* частина програми, що компілюється завжди */
#ifdef  DEBUG
     . . .                    /* компілюється тільки для налагодження */
   contr = ...;
   printf ("Контрольна точка 1: %d\n", contr);
#endif
```

Таких контрольних точок у програмі може бути декілька. Коли налагодження завершено, достатньо закоментувати директиву:

```
/* #define  DEBUG */
```

щоб відключити всі допоміжні засоби контролю.

Директиви `#ifdef` та `#ifndef` використовуються практично в усіх стандартних заголовних файлах *.h, зокрема для захисту програми від повторних включень заголовних файлів, які можуть виникнути через вкладення директив `#include`. Такий захист доцільно ввести і в користувацькі файли, що будуть приєднуватись до програм через `#include`. Наприклад, власний заголовний файл myheader.h можна заповнити так:

```
/*  Захист файла myheader.h від повторного включення */
#ifndef  _MY_HEADER_FILE_
#define  _MY_HEADER_FILE_
     . . .                    /* вміст файла */
#endif
```

У разі повторного підключення даного файла макрос `_MY_HEADER_FILE_` вже буде визначеним і директива `#ifndef` пропустить препроцесування вмісту файла.

Операція defined. Перевірити, чи ім'я заданого макроса визначене, можна також за допомогою операції `defined`, яка застосовується у виразах умови директив `#if` та `#elif` і записується так:

```
defined  ім'я_макроса
```

Якщо даний макрос був попередньо визначений, то результат операції `#defined` істинний (ненульовий), інакше результат перевірки визначеності хибний (нуль). Наприклад, перевірку визначеності макроса `DEBUG` можна організувати так:

```
#if  defined  DEBUG
     . . .                    /* засоби налагодження */
#endif
```

До результату операції `defined` можна застосовувати логічні операції `!`, `&&` та `||`. Зокрема перевірку невизначеністю макроса `_MY_HEADER_FILE_`, що становить умову приєднання до програми вмісту файла myheader.h, можна записати й так:

```
#if  !defined _MY_HEADER_FILE_
     . . .                          /* текст файла */
#endif
```

У наступному прикладі перевіряється визначеність обидвох заданих макросів:

```
#if  defined A && defined B
     . . .                    /* компілюється, коли визначено A та B */
#endif
```

Перевагою `defined` є те, що цю операцію можна застосовувати у виразі умови директиви `#elif` для організації груп послідовних перевірок.

17.4. Стандартні макроси

Стандарт С підтримує п'ять вбудованих (попередньо визначених) макросів, які дають змогу отримати певну інформацію на етапі препроцесування програми. Призначення їх таке:

- `__LINE__` – ціле значення, що дорівнює номеру рядка текстового файла програми, який опрацьовується в даний момент препроцесором (нумерація рядків починається з 1);
- `__FILE__` – символьний рядок, що містить ім'я файла програми, який в даний час опрацьовується препроцесором;
- `__DATE__` – символьний рядок, що містить дату початку компіляції програми у формі: "*Місяць день рік*";
- `__TIME__` – символьний рядок, що містить час початку компіляції програми у формі: "*година:хвилини:секунди*";
- `__STDC__` – набуває константного значення 1, якщо компілятор строго відповідає ANSI-стандарту мови (в Borland C такий режим компіляції можна встановити відповідною опцією меню OPTIONS), в інших випадках `__STDC__` вважається невизначеним.

Кожен з названих макросів починається і закінчується двома знаками підкреслення, щоб унеможливити їх випадкове перевизначення в програмі. Коли препроцесор зустрічає стандартний макрос, він підставляє замість нього відповідне значення. В усьому іншому ці макроси можна використовувати як звичайні користувацькі макроси.

Приклад звертання до вбудованих макросів для виведення інформації про хід компіляції програми.

```
/*  Звертання до стандартних макросів препроцесора */
    . . .                                  /* початкова частина програми */
    printf ("\n Опрацьовується %d-й рядок програми з файла %s \n"
            " Дата компіляції - %s, час початку - %s\n",
            __LINE__, __FILE__, __DATE__, __TIME__);
```

```
#if   defined __STDC__
     printf (" Компілятор працює за ANSI-стандартом.\n");
#endif
```

Результат препроцесування, що виводиться на екран у процесі виконання даного фрагмента програми, може бути, наприклад, таким:

```
Опрацьовується 24-й рядок програми з файла DEMOPREC.C
Дата компіляції - Mar 30 2004, час початку - 17:28:36
Компілятор працює за ANSI-стандартом.
```

17.5. Перегляд результатів препроцесування

Більшість компіляторів С не зупиняється після препроцесування і не створює окремого файла з результатами препроцесорного опрацювання програми. Щоб переглянути результати виконання директив препроцесора (така потреба виникає передусім, якщо є сумніви щодо правильності тексту, переданого на компіляцію), треба застосувати спеціальні команди (опції), властиві даному компілятору чи відповідному середовищу програмування.

До складу системи програмування Borland C входить автономна програма препроцесора cpp.exe, що дає змогу створити окремий файл з результатами препроцесування програми. Для цього достатньо вказати в командному рядку MS DOS:

cpp.exe *ім'я_файла_С-програми*

Після завершення роботи препроцесора буде створено текстовий файл з тим самим іменем і розширенням *.i, який міститиме текст програми з усіма виконаними директивами препроцесора. Переглянути цей файл можна будь-яким редактором чи переглядачем текстів.

Для прикладу наведемо результати препроцесування короткої демонстраційної програми, текст якої записано у файлі preprdem.c. Щоб вміст файла результатів препроцесування не був надто великим і незрозумілим, до програми не приєднується жодний зі стандартних заголовних файлів. Але оскільки в програмі відбувається звертання до бібліотечних функцій puts() та printf(), то замість стандартного заголовного файла stdio.h використано чотирирядковий користувацький файл включення output.h. У цьому файлі оголошено прототипи функцій puts() та printf() і вказано, що їх коди зберігаються у стандартній бібліотеці Borland C.

```
/*  Заголовний файл включення  output.h  */

extern "C" {
  int puts (const char far* st);
  int printf (const char far* format, ...);
}
```

Текст програми, записаної у файлі preprdem.c, що передається на опрацювання програмі-препроцесору cpp.exe, наступний:

```
#include "output.h"                          /* приєднання заголовного файла */
#define  ONE  100                                          /* макроконстанти */
#define  TWO  ONE+ONE
#define THREE(a)  3*a                                /* макрос з параметрами */
int main(void)
{
     int six1, six2;
   six1 = 6*ONE;          six2 = THREE(TWO);
#if  THREE(TWO) == 6*ONE                                  /* умовна компіляція */
   puts("Так: THREE(TWO) == 6*ONE");
#else
   puts ("Ні: THREE(TWO) != 6*ONE");
#endif
   printf("%d рядок - час: %s\n", __LINE__, __TIME__);
   return 0;
}
```

Нижче подано текст файла preprdem.i, отриманого в результаті препроцесування програми preprdem.c. Кожен рядок файла результатів починається іменем файла, з якого взято текст цього рядка, і номером рядка в базовому файлі. Ті рядки, в яких були записані директиви препроцесора, а також рядки, що не повинні компілюватись (визначені директивами умовної компіляції), залишаються порожніми. При цьому зберігається початкова нумерація рядків тексту програми, щоб на етапі компіляції у разі виявлення синтаксичних чи інших помилок можна було коректно вказати номер відповідного рядка. Усі коментарі з тексту програми вилучаються.

```
preprdem.c 1:
output.h 1:
output.h 2: extern "C" {
output.h 3:int puts(const char far * st);
output.h 4:int printf (const char far * format, ...);
output.h 5:}
output.h 6:
preprdem.c 2:
preprdem.c 3:
preprdem.c 4:
preprdem.c 5: int main(void)
preprdem.c 6: {
preprdem.c 7: int six1, six2;
preprdem.c 8: six1 = 6*100;          six2 = 3*100+100;
preprdem.c 9:
preprdem.c 10:
```

```
preprdem.c 11:
preprdem.c 12: puts ("Hi: THREE(TWO) != 6*ONE");
preprdem.c 13:
preprdem.c 14: printf("%d рядок - час: %s\n", 14, "13:06:12");
preprdem.c 15: return 0;
preprdem.c 16: }
preprdem.c 17:
```

Зверніть увагу на результати макропідстановок, виконаних через `#define`. Змінна `six1`, якій присвоєно значення виразу `6*ONE`, дорівнюватиме `6*100`, а змінна `six2`, яка набуває значення макроса `THREE(TWO)`, через відсутність дужок у виразі підстановки макроконстанти `TWO` дорівнюватиме `3*100+100`. Тому значення `six1` і `six2` будуть різними, що видно також із результатів виконання директиви `#if` – на компіляцію передається фрагмент програми, заданий у варіанті `#else`.

17.6. Інші директиви препроцесора

Наведені нижче директиви препроцесора використовуються порівняно рідко і мають допоміжний характер.

Директива `#line`. Вона призначена для зміни порядкового номера рядка та імені поточного файла в тексті результатів препроцесування. Синтаксис її такий:

```
#line  номер_рядка ім'я_файла
```

Константа *номер_рядка* вказує компілятору, що наступний рядок тексту в файлі результатів препроцесування повинен мати заданий номер. Якщо в цій директиві задано також *ім'я_файла* (його можна опустити), то в усіх наступних рядках поточний файл буде позначений заданим іменем.

Наприклад, якщо в тексті програми preprdem.c, поданому в попередньому параграфі, після рядка з номером 14 ввести директиву `#line`, а після неї вивести значення стандартних макросів:

```
#line 100 "newname.dem"
   printf("%d line - file: %s\n", __LINE__, __FILE__);
```

то останні рядки файла результатів препроцесування будуть такими:

```
    . . .
preprdem.c 14: printf("%d рядок - час: %s\n", 14, "13:06:12");
preprdem.c 15:
newname.dem 100: printf("%d рядок - файл: %s\n", 100, "newname.dem");
newname.dem 101: return 0;
newname.dem 102: }
newname.dem 103:
```

Директива `#error`. Вона призначена для виведення повідомлень про помилки на етапі препроцесування. Форма цієї директиви така:

```
#error  текст_повідомлення
```

Текст повідомлення записується без зовнішніх лапок. Директива виводить задане повідомлення про помилку і припиняє компіляцію програми.

Для прикладу змінимо в рядку 9 тексту програми preprdem.c умову компіляції та додатково введемо директиву `#error`:

```
#if  THREE(TWO) != 6*ONE
#error  Хибне визначення  TWO
```

Тоді процес компіляції програми буде перерваний на рядку 10 з виведенням екранного повідомлення:

```
Error preprdem.c 10: Error directive: Хибне визначення TWO
```

а в файл результатів препроцесування запишуться тільки 9 початкових рядків тексту програми.

Директива `#pragma`. Дії та можливості даної директиви не встановлені стандартом, вони визначаються конкретною реалізацією компілятора.

Система програмування Borland C підтримує ряд піддиректив директиви `#pragma`. Всі вони записуються у формі:

```
#pragma  ім'я_піддирективи  параметри
```

Частина піддиректив директиви `#pragma`, зокрема: `argused`, `hrdstop`, `inline`, `saveregs` не використовує параметрів.

Через директиву `#pragma` можна реалізовувати наступні дії:

- активізувати певні опції компілятора (подібно як при запуску з командного рядка) – піддиректива `option`;
- встановлювати функції, які будуть виконуватись на початку роботи програми (перед запуском `main()`) та в кінці (після `exit()`) – піддирективи `startup/exit`;
- керувати роботою `inline`-функцій – піддиректива `intrinsic`;
- зберігати вміст системних регістрів у разі виконання великих функцій – піддиректива `saveregs`;
- включати в текст програми оператори на мові асемблера – піддиректива `inline`;
- створювати окремий файл зі заголовних файлів, які вже пройшли препроцесування, – піддирективи `hrdfile/hrdstop`;
- відміняти повідомлення про невикористання оголошених параметрів програми – піддиректива `argused`.

Детальнішу інформацію щодо призначення піддиректив директиви `#pragma` та значень їх параметрів можна отримати із системи допомоги Help інтегрованого середовища Borland C.

Запитання та завдання для самоконтролю

1. Яке призначення директив препроцесора? На якому етапі виконання програми вони реалізуються?
2. Що виконує директива `#include`? Як записують ім'я файла включення? У яких випадках доцільно створювати власні заголовні файли?
3. За допомогою директиви `#define` створіть макрос (макропідстановку) `END` для позначення двох останніх рядків функцій:

```
    return 0;
}
```

4. Оголосіть дві макроконстанти: `PLUS` (зі значенням `1`) та `MINUS` (зі значенням `-1`) і напишіть макрос з параметром `SIGN(x)`, значенням якого буде знак аргументу (`PLUS` або `MINUS`). Наведіть приклад використання цього макроса.
5. Нехай для обчислення суми квадратів двох операндів у програмі записано макрос:

```
#define  SumSq(x1,x2)    x1*x1+x2*x2
```

   У яких випадках використання цього макроса може призвести до помилкового результату? Як виправити вираз підстановки, щоб макрос `SumSq()` став універсальним?
6. Яку роль у текстах макропідстановок виконує операція `#`? Що є її операндом, а що результатом?
7. Чи можна відмінити або змінити макровизначення у певній частині програми? Як це зробити?
8. Як забезпечити, щоб задана частина програми компілювалась тільки за певних умов? Які директиви використовують для організації умовної компіляції? Наведіть приклад фрагмента програми, який буде компілюватись тільки у випадку, коли значення макроконстанти `RESIST` дорівнює 2.
9. Заголовний файл програми, у якому записані функції керування мишкою, має таку структуру:

```
#ifndef  _MOUSE_
#define  _MOUSE_
     . . .                          /* тексти функцій */
#endif
```

   З якою метою в цьому файлі використано директиви `#ifndef` та `#define`?
10. Як переглянути результати препроцесорного опрацювання тексту програми в середовищі програмування Borland C? У якому файлі зберігається програма-препроцесор? Куди заносяться результати препроцесування?
11. Що додається препроцесором до тексту програми, а що – вилучається? З чого складається програмний файл, отриманий після препроцесорного опрацювання?
12. Що потрібно зробити, щоб на початку виконання програми на екран виводилась дата останньої компіляції тексту цієї програми?

Додаток 1

ТАБЛИЦІ КОДУВАННЯ СИМВОЛІВ

Таблиця Д1.1

ASCII-коди керуючих символів (коди 0 .. 31)

Позначення символа	10-й код	8-й код	16-й код	Призначення	Комбінація клавіш / клавіша
NUL	0	000	00	Нуль	^@
SOH	1	001	01	Початок заголовка	^A
STX	2	002	02	Початок тексту	^B
ETX	3	003	03	Кінець тексту	^C
EOT	4	004	04	Кінець передачі	^D
ENQ	5	005	05	Запит	^E
ACK	6	006	06	Підтвердження	^F
BEL	7	007	07	Сигнал (дзвінок)	^G
BS	8	010	08	Крок назад	^H / Bkspace
HT	9	011	09	Горизонтальна табуляція	^I / Tab
LF	10	012	0A	Переведення рядка	^J / Enter
VT	11	013	0B	Вертикальна табуляція	^K
FF	12	014	0C	Нова сторінка	^L
CR	13	015	0D	Повернення каретки	^M / Enter
SO	14	016	0E	Виключити зсунення	^N
SI	15	017	0F	Включити зсунення	^O
DLE	16	020	10	Ключ зв'язку даних	^P
DC1	17	021	11	Керування пристроєм 1	^Q
DC2	18	022	12	Керування пристроєм 2	^R
DC3	19	023	13	Керування пристроєм 3	^S
DC4	20	024	14	Керування пристроєм 4	^T
NAK	21	025	15	Негативне підтвердження	^U
SYN	22	026	16	Синхронізація	^V
ETB	23	027	17	Кінець блоку	^W
CAN	24	030	18	Відміна	^X
EM	25	031	19	Кінець середовища	^Y
SUB	26	032	1A	Заміна	^Z
ESC	27	033	1B	Ключ	^[/ Esc
FS	28	034	1C	Роздільник файлів	^\
GS	29	035	1D	Роздільник групи	^]
RS	30	036	1E	Роздільник записів	^^
US	31	037	1F	Роздільник елементів	^_

Примітка:
^@, ^A і т. д. позначає комбінацію клавіш: *Ctrl*+@, *Ctrl*+*A* і т. д.

ASCII-коди графічних символів (коди 32.. 127)

Символ	10-й код	8-й код	16-й код	Символ	10-й код	8-й код	16-й код
пробіл	32	40	20	F	70	106	46
!	33	41	21	G	71	107	47
"	34	42	22	H	72	110	48
#	35	43	23	I	73	111	49
$	36	44	24	J	74	112	4A
%	37	45	25	K	75	113	4B
&	38	46	26	L	76	114	4C
'	39	47	27	M	77	115	4D
(	40	50	28	N	78	i16	4E
)	41	51	29	O	79	117	4F
*	42	52	2A	P	80	120	50
+	43	53	2B	Q	81	121	51
,	44	54	2C	R	82	122	52
-	45	55	2D	S	83	123	53
.	46	56	2E	T	84	124	54
/	47	57	2F	U	85	125	55
0	48	60	30	V	86	126	56
1	49	61	31	W	87	127	57
2	50	62	32	X	88	130	58
3	51	63	33	Y	89	131	59
4	52	64	34	Z	90	132	5A
5	53	65	35	[	91	133	5B
6	54	66	36	\	92	134	5C
7	55	67	37	]	93	135	5D
8	56	70	38	^	94	136	5E
9	57	71	39	_	95	137	5F
:	58	72	3A	`	96	140	60
;	59	73	3B	a	97	141	61
<	60	74	3C	b	98	142	62
=	61	75	3D	c	99	143	63
>	62	76	3E	d	100	144	64
?	63	77	3F	e	101	145	65
@	64	100	40	f	102	146	66
A	65	101	41	g	103	147	67
B	66	102	42	h	104	150	68
C	67	103	43	i	105	151	69
D	68	104	44	j	106	152	6A
E	69	105	45	k	107	153	6B

Закінчення табл. Д1.2

Символ	10-й код	8-й код	16-й код	Символ	10-й код	8-й код	16-й код
l	108	154	6C	v	118	166	76
m	109	155	6D	w	119	167	77
n	110	156	6E	x	120	170	78
o	111	157	6F	y	121	171	79
p	112	160	70	z	122	172	7A
q	113	161	71	{	123	173	7B
r	114	162	72	|	124	174	7C
s	115	163	73	}	125	175	7D
t	116	164	74	~	126	176	7E
u	117	165	75	*Del*	127	177	7F

Таблиця Д1.3

Розширена кодова таблиця MS DOS (коди 128.. 255)

Символ	10-й код	8-й код	16-й код	Символ	10-й код	8-й код	16-й код
А	128	200	80	Ш	152	230	98
Б	129	201	81	Щ	153	231	99
В	130	202	82	Ъ	154	232	9A
Г	131	203	83	Ы	155	233	9B
Д	132	204	84	Ь	156	234	9C
Е	133	205	85	Э	157	235	9D
Ж	134	206	86	Ю	158	236	9E
З	135	207	87	Я	159	237	9F
И	136	210	88	а	160	240	A0
Й	137	211	89	б	161	241	A1
К	138	212	8A	в	162	242	A2
Л	139	213	8B	г	163	243	A3
М	140	214	8C	д	164	244	A4
Н	141	215	8D	е	165	245	A5
О	142	216	8E	ж	166	246	A6
П	143	217	8F	з	167	247	A7
Р	144	220	90	и	168	250	A8
С	145	221	91	й	169	251	A9
Т	146	222	92	к	170	252	AA
У	147	223	93	л	171	253	AB
Й	148	224	94	м	172	254	AC
Х	149	225	95	н	173	255	AD
Ц	150	226	96	о	174	256	AE
Ч	151	227	97	п	175	257	AF

Символ	10-й код	8-й код	16-й код	Символ	10-й код	8-й код	16-й код
▒	176	260	B0	╪	216	330	D8
▒	177	261	B1	┘	217	331	D9
▒	178	262	B2	┌	218	332	DA
│	179	263	B3	▒	219	333	DB
┤	180	264	B4	▒	220	334	DC
╡	181	265	B5	▒	221	335	DD
╢	182	266	B6	▒	222	336	DE
╖	183	267	B7	▒	223	337	DF
╕	184	270	B8	р	224	340	E0
╣	185	271	B9	с	225	341	E1
║	186	272	BA	т	226	342	E2
╗	187	273	BB	у	227	343	E3
╝	188	274	BC	ф	228	344	E4
╜	189	275	BD	х	229	345	E5
╛	190	276	BE	ц	230	346	E6
┐	191	277	BF	ч	231	347	E7
└	192	300	C0	ш	232	350	E8
┴	193	301	C1	щ	233	351	E9
┬	194	302	C2	ъ	234	352	EA
├	195	303	C3	ы	235	353	EB
─	196	304	C4	ь	236	354	EC
┼	197	305	C5	э	237	355	ED
╞	198	306	C6	ю	238	356	EE
╟	199	307	C7	я	239	357	EF
╚	200	310	C8	≡ / Ё	240	360	F0
╔	201	311	C9	± / ё	241	361	F1
╩	202	312	CA	Ґ	242	362	F2
╦	203	313	CB	ґ	243	363	F3
╠	204	314	CC	Є	244	364	F4
═	205	315	CD	є	245	365	F5
╬	206	316	CE	І	246	366	F6
╧	207	317	CF	і	247	367	F7
╨	208	320	D0	Ї	248	370	F8
╤	209	321	D1	ї	249	371	F9
╥	210	322	D2	·	250	372	FA
╙	211	323	D3	√	251	373	FB
╘	212	324	D4	ⁿ	252	374	FC
╒	213	325	D5	²	253	375	FD
╓	214	326	D6	■	254	376	FE
╫	215	327	D7		255	377	FF

Розширена кодова таблиця MS Windows (коди 128.. 255)

Символ	10-й код	8-й код	16-й код	Символ	10-й код	8-й код	16-й код
Ђ	128	200	80	¦	166	246	A6
Ѓ	129	201	81	§	167	247	A7
‚	130	202	82	Ё	168	250	A8
ѓ	131	203	83	©	169	251	A9
„	132	204	84	Є	170	252	AA
…	133	205	85	«	171	253	AB
†	134	206	86	¬	172	254	AC
‡	135	207	87	–	173	255	AD
€	136	210	88	®	174	256	AE
‰	137	211	89	Ї	175	257	AF
Љ	138	212	8A	°	176	260	B0
‹	139	213	8B	±	177	261	B1
Њ	140	214	8C	І	178	262	B2
Ќ	141	215	8D	і	179	263	B3
Ћ	142	216	8E	ґ	180	264	B4
Џ	143	217	8F	µ	181	265	B5
ђ	144	220	90	¶	182	266	B6
‘	145	221	91	·	183	267	B7
’	146	222	92	ё	184	270	B8
“	147	223	93	№	185	271	B9
”	148	224	94	є	186	272	BA
•	149	225	95	»	187	273	BB
–	150	226	96	j	188	274	BC
—	151	227	97	S	189	275	BD
□	152	230	98	s	190	276	BE
™	153	231	99	ї	191	277	BF
љ	154	232	9A	А	192	300	C0
›	155	233	9B	Б	193	301	C1
њ	156	234	9C	В	194	302	C2
ќ	157	235	9D	Г	195	303	C3
ћ	158	236	9E	Д	196	304	C4
џ	159	237	9F	Е	197	305	C5
	160	240	A0	Ж	198	306	C6
Ў	161	241	A1	З	199	307	C7
ў	162	242	A2	И	200	310	C8
J	163	243	A3	Й	201	311	C9
¤	164	244	A4	К	202	312	CA
Ґ	165	245	A5	Л	203	313	CB

Символ	10-й код	8-й код	16-й код	Символ	10-й код	8-й код	16-й код
М	204	314	CC	ж	230	346	E6
Н	205	315	CD	з	231	347	E7
О	206	316	CE	и	232	350	E8
П	207	317	CF	й	233	351	E9
Р	208	320	D0	к	234	352	EA
С	209	321	D1	л	235	353	EB
Т	210	322	D2	м	236	354	EC
У	211	323	D3	н	237	355	ED
Ф	212	324	D4	о	238	356	EE
Х	213	325	D5	п	239	357	EF
Ц	214	326	D6	р	240	360	F0
Ч	215	327	D7	с	241	361	F1
Ш	216	330	D8	т	242	362	F2
Щ	217	331	D9	у	243	363	F3
Ъ	218	332	DA	ф	244	364	F4
Ы	219	333	DB	х	245	365	F5
Ь	220	334	DC	ц	246	366	F6
Э	221	335	DD	ч	247	367	F7
Ю	222	336	DE	ш	248	370	F8
Я	223	337	DF	щ	249	371	F9
а	224	340	E0	ъ	250	372	FA
б	225	341	E1	ы	251	373	FB
в	226	342	E2	ь	252	374	FC
г	227	343	E3	э	253	375	FD
д	228	344	E4	ю	254	376	FE
е	229	345	E5	я	255	377	FF

Додаток 2

ФУНКЦІЇ БІБЛІОТЕКИ BORLAND C

У таблицях Д2.1 – Д2.7 подано перелік бібліотечних функцій Borland C, оголошених у заголовних файлах: `<math.h>`, `<ctype.h>`, `<string.h>`, `<stdlib.h>`, `<stdio.h>`, `<time.h>` та `<conio.h>`. Для кожної функції наведено прототип, вказано призначення всіх параметрів, описано дії, які реалізує ця функція, та значення, які вона повертає за різних умов завершення. У примітках до таблиць розкрито макроконстанти та спеціальні іменовані типи, що застосовуються у відповідних функціях.

Таблиця Д2.1

Математичні функції (заголовний файл `<math.h>`)

Функція	Прототип і призначення
`abs()`	`int abs (int a);` Повертає абсолютне значення цілочислового параметра `a` ($\|a\|$).
`acos()`	`double acos (double x);` Обчислює і повертає значення арккосинуса параметра `x` (*arccos x*) у діапазоні [0, π]. Значення `x` має потрапляти в межі від `-1` до `1`, інакше виводиться повідомлення `NAN` (Not A Number – не число) і встановлюється код помилки `EDOM` (див. прим. 1).
`asin()`	`double asin (double x);` Обчислює і повертає значення арксинуса параметра `x` (*arcsin x*) у діапазоні [-π/2, π/2]. Значення `x` має потрапляти в межі від `-1` до `1`, інакше виводиться повідомлення `NAN` і встановлюється код помилки `EDOM`.
`atan()`	`double atan (double x);` Обчислює і повертає значення арктангенса параметра `x` (*arctg x*) у діапазоні [-π/2, π/2].
`atan2()`	`double atan2 (double x, double y);` Обчислює і повертає значення арктангенса частки `x/y` (*arctg x/y*) у діапазоні [0, π]. Якщо обидва параметри дорівнюють `0`, то встановлює код помилки `EDOM`.
`cabs()`	`double cabs (struct complex z);` Повертає абсолютне значення комплексного параметра `z` ($\|z\|$), дійсна та уявна частини якого записані в поля структури, що має тип `struct complex` (див. прим. 4).
`ceil()`	`double ceil (double x);` Повертає дійсне значення найменшого цілого числа, що не менше за `x` ($\geq x$).
`cos()`	`double cos (double x);` Обчислює і повертає значення косинуса параметра `x` (*cos x*) у діапазоні від `-1` до `1`. Значення `x` має бути задане в радіанах.

Функція	Прототип і призначення
`cosh()`	`double cosh (double x);` Обчислює і повертає значення гіперболічного косинуса параметра `x` ($ch\,x = (e^x + e^{-x})/2$). Якщо результат викликає переповнення, то повертає `HUGE_VAL` і встановлює код помилки `ERANGE` (див. прим. 2, 3).
`exp()`	`double exp (double x);` Обчислює і повертає значення експоненти параметра `x` (e^x). У разі переповнення повертає `HUGE_VAL` і встановлює код помилки `ERANGE`.
`fabs()`	`double fabs (double x);` Повертає абсолютне значення дійсного параметра `x` ($\lvert x \rvert$).
`floor()`	`double floor (double x);` Повертає дійсне значення найбільшого цілого числа, що не більше за `x` ($\le x$).
`fmod()`	`double fmod (double x double y);` Обчислює і повертає дійсне значення (g), що дорівнює остачі від ділення параметрів `x` та `y` ($x = ky + g$, k – ціле число, $0 < g < y$).
`frexp()`	`double frexp (double x, int * pexp);` Виділяє нормалізовану мантису (m) і порядок (p) дійсного значення параметра `x` ($x = m \cdot 2^p$, k – ціле число, $0 \le m < 1$). Повертає значення мантиси, а значення порядку записує за адресою, яку задає вказівник `pexp`.
`hypot()`	`double hypot (double x, double y);` Обчислює і повертає значення гіпотенузи прямокутного трикутника, катети якого задаються параметрами `x` та `y` ($\sqrt{x^2+y^2}$). У разі переповнення повертає `HUGE_VAL` і встановлює код помилки `ERANGE`.
`labs()`	`long int labs (long int b);` Повертає абсолютне значення параметра `b`, що має тип `long int` ($\lvert b \rvert$).
`ldexp()`	`double ldexp (double x, int p);` Обчислює і повертає значення виразу $x \cdot 2^p$, де параметр `p` цілочисловий.
`log()`	`double log (double x);` Обчислює і повертає значення натурального логарифма параметра `x` ($ln\,x$). Значення `x` має бути дійсним додатним числом: `x > 0`. У противному випадку, якщо `x < 0`, то встановлює код помилки `EDOM`, а якщо `x = 0`, то код помилки `ERANGE`.
`log10()`	`double log10 (double x);` Обчислює і повертає значення десяткового логарифма параметра `x` ($lg\,x$). Значення `x` має бути дійсним додатним числом: `x > 0`. У противному випадку, якщо `x < 0`, то встановлює код помилки `EDOM`, а якщо `x = 0`, то код помилки `ERANGE`.
`modf()`	`double modf (double x, double * ipart);` Розбиває дійсне значення параметра `x` на цілу та дробову частини. Повертає дробову частину числа, а цілу записує за адресою, яку задає вказівник `ipart`.
`poly()`	`double poly (double x, int n, double c[]);` Обчислює і повертає значення полінома параметра `x` ($c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n$). Параметр `n` задає порядок полінома, а параметр `c` – адресу масиву, де записані коефіцієнти полінома.

Функція	Прототип і призначення
`pow()`	`double pow (double x, double y);` Обчислює і повертає результат піднесення параметра `x` до дійсного степеня `y` (x^y). Фіксує код помилки `EDOM`, якщо: 1) $x=0$, а $y\le 0$; 2) $x<0$, а `y` – не є цілим. У разі переповнення повертає `HUGE_VAL` і встановлює код помилки `ERANGE`.
`pow10()`	`double pow10 (int p);` Обчислює і повертає як дійсне число результат піднесення основи `10` до цілого степеня `p` (10^p). У разі переповнення повертає `HUGE_VAL` і встановлює код помилки `ERANGE`.
`sin()`	`double sin (double x);` Обчислює і повертає значення синуса параметра `x` ($\sin x$) у діапазоні від `-1` до `1`. Значення `x` має бути задане в радіанах.
`sinh()`	`double sinh (double x);` Обчислює і повертає значення гіперболічного синуса параметра `x` ($sh\,x = (e^x - e^{-x})/2$). Якщо результат викликає переповнення, то повертає `HUGE_VAL` і встановлює код помилки `ERANGE` (див. прим. 2, 3).
`sqrt()`	`double sqrt (double x);` Обчислює і повертає значення кореня квадратного з параметра `x` ($\sqrt{x}$). Значення `x` має бути дійсним додатним числом ($x \ge 0$). Інакше встановлює код помилки `EDOM`.
`tan()`	`double tan (double x);` Обчислює і повертає значення тангенса параметра `x` ($tg\,x$). Значення `x` має бути задане в радіанах.
`tanh()`	`double tanh(double x);` Обчислює і повертає значення гіперболічного тангенса параметра `x` ($th\,x = sh/ch$).

Примітки:

1. `EDOM` – макроконстанта, що сигналізує про загальну помилку результату виконання математичної операції (Error Domain). У Borland C її значення дорівнює `33`.
2. `ERANGE` – макроконстанта, що сигналізує про помилку переповнення діапазону в процесі виконання математичної операції (Out of Range Error). У Borland C її значення дорівнює `34`.
3. `HUGE_VAL` – значення, яке повертають математичні функції, коли результат виходить за межі, допустимі для типу `double`. Збігається з найбільшим можливим значенням цього типу.
4. `struct complex` – структура, в поля якої записуються значення дійсної та уявної частин комплексного числа. Її шаблон оголошено так:

```
struct complex {
   double x;          /* дійсна частина */
   double y;          /* уявна частина  */
};
```

5. `M_PI`, `M_PI2`, ... – макроконстанти значень константи π (3,1415926535...) та декількох математичних виразів із π (оголошені з точністю 21-ї десяткової цифри):

`M_PI` – π `M_PI2` – $\pi/2$ `M_PI4` – $\pi/4$
`M_1_PI` – $1/\pi$ `M_2_PI` – $2/\pi$
`M_1_SQRTPI` – $1/\sqrt{\pi}$ `M_2_SQRTPI` – $2/\sqrt{\pi}$

6. `M_E`, `M_LOG2E`, ... – макроконстанти значень константи e (2.718281828...) та деяких математичних функцій, пов'язаних із e (оголошені з точністю 21-ї десяткової цифри):
   `M_E` – e
   `M_LOG2E` – $log_2 e$ `M_LOG10E` – $lg e$
   `M_LN2` – $ln 2$ `M_LN10` – $ln 10$
7. Бібліотека системи програмування Borland C/C++ додатково містить набір математичних функцій, ідентичних до поданих у табл. Д2.1, але призначених для роботи з даними, що мають тип `long double`. Імена цих функцій закінчуються літерою `l` (наприклад, `cosl()`, `hypotl()`, `sqrtl()`), а всі дійсні параметри та значення мають тип `long double`.

Таблиця Д2.2

Функції класифікації та зміни символів (заголовний файл `<ctype.h>`)

Функція	Прототип і призначення
`isalnum()`	`int isalnum (int sym);` Повертає ненульове значення, якщо символ `sym` є літерою або цифрою (`'A'..'Z'`, `'a'..'z'` або `'0'..'9'`), інакше повертає нуль.
`isalpha()`	`int isalpha (int sym);` Повертає ненульове значення, якщо символ `sym` є латинською літерою (`'A'..'Z'` або `'a'..'z'`), інакше – нуль.
`isascii()`	`int isascii (int sym);` Повертає ненульове значення, якщо символ `sym` входить у стандартну частину ASCII-таблиці (коди 0..127), інакше – нуль.
`iscntrl()`	`int iscntrl (int sym);` Повертає ненульове значення, якщо `sym` належить до групи керуючих символів (коди 0..31 або 127), інакше – нуль.
`isdigit()`	`int isdigit (int sym);` Повертає ненульове значення, якщо символ `sym` є десятковою цифрою (`'0'..'9'`), інакше – нуль.
`isgraph()`	`int isgraph (int sym);` Повертає ненульове значення, якщо `sym` належить до друкованих символів, крім пробілу (коди 33..126), інакше – нуль.
`islower()`	`int islower(int sym);` Повертає ненульове значення, якщо символ `sym` є малою латинською літерою (`'a'..'z'`), інакше – нуль.
`isprint()`	`int isprint (int sym);` Повертає ненульове значення, якщо `sym` належить до групи друкованих символів (коди 32..126), інакше – нуль.
`ispunct()`	`int ispunct (int sym);` Повертає ненульове значення, якщо `sym` належить до символів пунктуації (`iscntrl(sym)` або `isspace(sym)`), інакше – нуль.
`isspace()`	`int isspace (int sym);` Повертає ненульове значення, якщо символ `sym` є пробільним символом (символом пробілу, нового рядка, нової сторінки, горизонтальної чи вертикальної табуляції – коди 9..13 або 32), інакше – нуль.

Функція	Прототип і призначення
`isupper()`	`int isupper (int sym);` Повертає ненульове значення, якщо символ `sym` є великою латинською літерою (`'A'..'Z'`), інакше – нуль.
`isxdigit()`	`int isxdigit (int sym);` Повертає ненульове значення, якщо `sym` є символом шістнадцяткових цифр (`'0'..'9'`, `'A'..'F'` або `'a'..'f'`), інакше – нуль.
`toascii()`	`int toascii (int sym);` Повертає `sym%128` - перетворює значення `sym` до ASCII-коду.
`tolower()`	`int tolower(int sym);` Повертає відповідну малу літеру, якщо `sym` велика латинська літера, інакше – `sym`.
`toupper()`	`int toupper (int sym);` Повертає відповідну велику літеру, якщо `sym` мала латинська літера, інакше – `sym`.

Таблиця Д2.3

Функції для роботи з ділянками оперативної пам'яті (ОП) і символьними рядками (заголовний файл `<string.h>`)

Функція	Прототип і призначення
`memccpy()`	`void* memccpy (void* m1, const void* m2, int sym, size_t n);` Аналог `memcpy()`, але зупиняє копіювання байтів з ділянки `m2`, якщо в `m1` скопійовано символ `sym`. Повертає адресу байта, наступного за байтом із символом `sym` у `m1`, або `NULL`, якщо `sym` не скопійовано.
`memchr()`	`void* memchr (const void* m, int sym, size_t n);` Перевіряє, чи серед `n` перших байтів ділянки ОП, заданої адресою `m`, є байт, що дорівнює коду символа `sym`. Повертає адресу цього байта або `NULL`, якщо `sym` не зустрічається у заданій ділянці.
`memcmp()`	`int memcmp (const void* m1, const void* m2, size_t n);` Попарно порівнює (як дані з типом `unsigned char`) `n` перших байтів ділянок ОП, заданих вказівниками `m1` і `m2`. Повертає значення цілого типу: 0, якщо вміст `m1` збігається з `m2`; <0, якщо `m1<m2`; >0, якщо `m1>m2`.
`memcpy()`	`void* memccpy (void* m1, const void* m2, size_t n);` Копіює `n` перших байтів з ділянки ОП `m2` у ділянку ОП `m1` (ділянки не повинні перекриватись). Повертає `m1` – адресу скопійованої ділянки.
`memicmp()`	`int memicmp (const void* m1, const void* m2, size_t n);` Аналог `memcmp()`, але порівнює байти як символи, ігноруючи їх регістри (великі й малі літери не розрізняються).
`memmove()`	`void* memmove(void* m1, const void* m2, size_t n);` Аналог `memcpy()`, але додатково дозволяє, щоб ділянки ОП `m1` та `m2` перекривались.

Функція	Прототип і призначення
`memset()`	`void * memset (void * m, int sym, size_t n);` Записує у перші `n` байт ділянки ОП, адресу якої задає параметр `m`, код символа `sym`.
`movedata()`	`void movedata (unsigned m1seg, unsigned m1off, unsigned m2segm2, unsigned m2off, size_t n);` Копіює `n` байт з ділянки ОП, заданої адресою `m1seg:m1off` у ділянку `m2seg:m2off` (сегмент:зміщення). Задання повних адрес робить функцію незалежною від моделі пам'яті.
`stpcpy()`	`char * stpcpy (char * s1, const char * s2);` Копіює рядок `s2` (з `'\0'` включно) за адресою, заданою через `s1`. Повертає адресу кінця скопійованого рядка – значення `s1+strlen(s2)`.
`strcat()`	`char * strcat (char * s1, const char * s2);` Приєднує рядок `s2` (з `'\0'` включно) до кінця рядка `s1`. Повертає адресу доповненого рядка – значення `s1` .
`strchr()`	`char * strchr (const char * s, int sym);` Перевіряє, чи символ `sym` входить у рядок `s`. Повертає вказівник на перше входження `sym` у `s` або `NULL`, якщо `sym` не зустрічається в рядку `s`.
`strcmp()`	`int strcmp (const char * s1, const char * s2);` Послідовно порівнює символи рядків `s1` і `s2` як дані `unsigned char`. Повертає ціле число, значення якого: `<0`, якщо `s1<s2`; `0`, якщо `s1` і `s2` збігаються; `>0`, якщо `s1>s2`.
`strcmpi()`	`int strcmpi (const char * s1, const char * s2);` Макрос, що викликає функцію `stricmp()`.
`strcpy()`	`char * strcpy (char * s1, const char * s2);` Копіює рядок `s2` (з `'\0'` включно) за адресою, заданою через `s1`. Повертає `s1` – адресу скопійованого рядка.
`strcspn()`	`size_t strcspn (const char * s1, const char * s2);` Повертає кількість початкових символів рядка `s1`, які не входять у рядок `s2` (довжину підрядка `s1` до появи першого символа з `s2`).
`strdup()`	`char * strdup (const char * s);` Копіює рядок `s` (з `'\0'` включно) в динамічну пам'ять, попередньо виділивши там ділянку розміром `strlen(s)+1`. Повертає адресу рядка в динамічній пам'яті.
`strerror()`	`char * strerror (int errnmb);` Повертає вказівник на початок рядка зі стандартним повідомленням про помилку процесу виконання програми з номером `errnmb`.
`stricmp()`	`int stricmp (const char * s1, const char * s2);` Аналог `strcmp()`, але порівнює символи рядків `s1` і `s2`, ігноруючи їх регістри (великі й малі літери не розрізняються).
`strlen()`	`unsigned strlen (const char s);` Повертає довжину рядка `s` у символах (`'\0'` не враховується).
`strlwr()`	`char * strlwr (char * s);` Замінює великі латинські літери у рядку `s` відповідними малими літерами, інші символи не змінюються. Повертає `s` – адресу перетвореного рядка.

Функція	Прототип і призначення
`strncat()`	`char* strncat (char* s1, const char* s2, size_t n);` Аналог `strcat()`, але долучає до рядка `s1` не більше, ніж `n` початкових символів з рядка `s2`; у кінець доповненого рядка заноситься `'\0'`.
`strncmp()`	`char* strncmp (const char* s1, const char* s2, size_t n);` Аналог `strcmp()`, але порівнює тільки `n` початкових символів рядків `s1` та `s2`. Якщо якийсь із рядків коротший за `n`, то порівняння припиняється з досягненням `'\0'`.
`strncmpi()`	`char* strncmpi (const char* s1, const char* s2, size_t n);` Аналог `strncmp()`, але порівнює символи рядків `s1` і `s2`, ігноруючи їх регістри (великі й малі літери не розрізняються).
`strncpy()`	`char* strncpy (char* s1, const char* s2, int n);` Аналог `strcpy()`, але з `s2` у рядок `s1` копіюється не більше, ніж `n` початкових символів. Якщо скопійована група символів не закінчується `'\0'`, то нуль-символ у `s1` не заноситься.
`strnicmp()`	`char* strnicmp (const char* s1, const char* s2, size_t n);` Макрос, що викликає функцію `strncmpi()`.
`strnset()`	`char* strnset (char* s, int sym, size_t n);` Аналог `strset()`, але у рядок `s` заноситься не більше, ніж `n` символів `sym`. Заповнення рядка `s` припиняється, якщо досягнуто `'\0'`.
`strpbrk()`	`char* strpbrk (const char* s1, const char* s2);` Шукає у рядку `s1` перший символ, що входить у рядок `s2`. Повертає вказівник на знайдений символ або `NULL`, якщо в рядку `s1` немає символів, з яких складається рядок `s2`.
`strrchr()`	`char* strrchr (const char* s, int sym);` Аналог `strchr()`. Повертає вказівник на останнє входження символа `sym` у рядок `s` або `NULL`, якщо `sym` не зустрічається в `s`.
`strrev()`	`char* strrev (char* s);` Переставляє символи рядка `s` (до `'\0'`) у зворотному порядку. Повертає `s` – адресу рядка після реверсування.
`strset()`	`char* strset (char* s, int sym);` Заповнює рядок `s` (до `'\0'`) символом `sym`. Повертає `s` – адресу заповненого рядка.
`strspn()`	`size_t strspn (const char* s1, const char* s2);` Повертає кількість початкових символів рядка `s1`, які входять у рядок `s2` (довжину початкового підрядка, що складається із символів `s2`).
`strstr()`	`char* strstr (const char* s1, const char* s2);` Перевіряє, чи рядок `s2` входить як підрядок у `s1`. Повертає вказівник на перший символ рядка `s2` у `s1` або `NULL`, якщо `s2` не зустрічається у рядку `s1`.
`strtok()`	`char* strtok (char* s1, const char* s2);` Виділяє в рядку `s1` лексеми, обмежені символами з рядка `s2` (див. §9.4.2). Повертає вказівник на виділену лексему або `NULL`.

Функція	Прототип і призначення
`strupr()`	`char* strupr (char* s);` Замінює малі латинські літери у рядку `s` відповідними великими літерами, інші символи не змінюються. Повертає `s` – адресу перетвореного рядка.
`_strerror()`	`char* _strerror (const char* errtext);` Формує символьний рядок із повідомленням про останню зафіксовану помилку процесу виконання програми у формі `errtext: стандартне повідомлення`. Повертає вказівник на початок сформованого рядка (рядок записується у спеціальний внутрішній статичний буфер).

Примітки:

1. `size_t` – тип, що використовується для мобільності параметрів розміру даних і обсягу ділянок пам'яті. Задекларований через `typedef` як тип `unsigned int`.
2. `NULL` – макроконстанта, що позначає порожній вказівник. Її значення дорівнює `0`.

Таблиця Д2.4

Функції загального призначення
(заголовний файл <stdlib.h>)

Функція	Прототип і призначення
`abort()`	`void abort(void);` Зупиняє виконання програми (аварійне завершення). Виводить у потік помилок повідомлення `"Abnormal program termination"` і викликає функцію завершення `_exit()` з кодом 3.
`abs()`	`int abs (int number);` Повертає абсолютне значення аргументу `number`, який має тип `int`.
`atexit()`	`int atexit (void (*fcmp)(void ));` Реєструє функцію, адресу якої задає параметр `func`, як функцію, що буде автоматично виконуватись перед кожним неаварійним завершенням програми. Можна зареєструвати до 32-х функцій завершення. Порядок виклику функцій завершення зворотний до порядку їх реєстрації (першою виконується функція, зареєстрована останньою, і т.д.).
`atof()`	`double atof (const char* st);` Виділяє зі символьного рядка `st` підрядок, який є записом дійсного числа, і перетворює його в значення з типом `double`. Перед числом у `st` може бути довільна кількість символів роздільників. Саме число повинно бути записане як десяткове ціле чи дійсне у формі з фіксованою або з плаваючою крапкою. Кінцем рядка вважається перший символ, який не відноситься до символів дійсного числа. У разі успішного перетворення функція повертає обчислене числове значення; якщо число виділити не вдалось, то функція повертає 0; а в разі виникнення переповнення – значення `±HUGE_VAL` і фіксує помилку переповнення.
`_atold()`	`long double _atold (const char* st);` Аналог `atof()`, але перетворює число із символьного рядка `st` у дане з типом `long double`.

Функція	Прототип і призначення
`atoi()`	`int atoi (const char* st);` Виділяє із символьного рядка `st` підрядок, який відповідає цілому числу, і перетворює його у значення з типом `int`. Числу в `st` може передувати довільна кількість символів роздільників. Саме число повинно бути записане як десяткове ціле, перед числом може бути знак `+` або `-`. Кінцем числа вважається перший нецифровий символ. У разі успішного перетворення функція повертає обчислене значення, а в разі помилки – `0`.
`_atol()`	`long atol (const char*st);` Аналог `atoi()`, але перетворює вміст символьного рядка `st` у число з типом `long int`.
`bsearch()`	`void* bsearch (const void* key, const void* arr, size_t nelem, size_t size, int (*fcmp)(const void*, const void*));` Виконує двійковий пошук у впорядкованому масиві даних, адресу початку якого задає параметр `arr`. Повертає адресу першого елемента `arr`, який збігається із ключем пошуку, заданим вказівником `key`, або `NULL`, якщо відповідного елемента в масиві немає. Параметр `nelem` задає кількість елементів масиву, а параметр `size` – розмір елемента в байтах (див. прим. 2). Пошук виконується за методом половинного ділення, елементи `arr` повинні бути впорядковані за зростанням значень. Для порівняння елементів масиву із ключовим значенням `*key` використовується окрема функція, адресу якої задає останній параметр `fcmp`. Функція порівняння повинна повертати значення `<0`, якщо її перший аргумент менший за другий, `0`, якщо аргументи збігаються, і значення `>0`, якщо перший аргумент більший за другий.
`calloc()`	`void* calloc (size_t num, size_t size);` Виділяє неперервну ділянку динамічної пам'яті, достатню для розташування масиву з `num` елементів, кожен з яких має розмір `size` (див. прим. 2). Виділена ділянка заповнюється нульовими байтами (онулюється). Місце розташування динамічної ділянки залежить від встановленої моделі пам'яті (див. §14.5). У разі успішного виконання повертає адресу першого байта виділеної ділянки, а в разі невдачі – `NULL`.
`div()`	`div_t div (int numer, int denom);` Виконує ділення двох цілих чисел: `numer` – ділене, `denom` – дільник. Повертає структуру з типом `div_t` (див. прим. 3), два цілочислових поля якої заповнюються часткою (поле `quot`) і залишком (поле `rem`) від ділення.
`ecvt()`	`char* ecvt (double numb, int ndig, int* dec, int* sign);` Перетворює дійсне числове значення, задане параметром `numb`, у внутрішній символьний рядок, що складається із `ndig` десяткових цифр числа. Повертає адресу першого байта сформованого рядка (кожен наступний виклик `ecvt()` змінює вміст рядка). За адресою, заданою через `dec`, записує позицію десяткової крапки відносно початку рядка (сам рядок крапки не містить). Якщо `*dec < 0`, то десяткова крапка розташована зліва, тобто число `<1`. Вказівник `sign` задає адресу, за якою записується знак числа: `0`, якщо `numb` додатне число, і ненульове значення, якщо `numb` від'ємне.
`exit()`	`void exit (int status);` Завершує виконання програми. Попередньо записує у файли вміст відповідних буферів виведення, закриває всі відкриті файли, викликає функції, зареєстровані як функції завершення (див. `atexit()`). Параметр `status`

Функція	Прототип і призначення
	передає в операційну систему статус завершення програми. Прийнято, що 0 (або EXIT_SUCCESS) вказує на нормальне завершення, а ненульове значення (або EXIT_FAILURE) є ознакою аварійного завершення.
_exit()	`void _exit (int status);` Аналог функції exit(), але не виконує скидання буферів, не закриває файли даних і не викликає функцій завершення.
fcvt()	`char* fcvt (double numb, int ndig, int* dec, int* sign);` Аналог ecvt(), але параметр ndig задає кількість цифр у дробовій частині числа, записаного в символьний рядок.
free()	`void free (void* memp);` Звільняє ділянку динамічної пам'яті, попередньо виділену однією із функцій: calloc(), malloc() чи realloc(). Параметр memp повинен задавати адресу першого байта ділянки, що звільняється.
_fullpath()	`char* _fullpath (char* buf, const char *path, int buflen);` Формує і заносить у буфер buf символьний рядок, що задає повний шлях до файла з урахуванням відносного шляху path і поточного активного каталога. Параметр buflen задає розмір буфера buf. Повертає адресу buf у разі успішного виконання та NULL, якщо буфер недостатнього розміру чи шлях path містить помилку.
gcvt()	`char* gcvt (double numb, int ndec, char* buf);` Перетворює дійсне значення параметра numb у символьний рядок і записує його за адресою buf. Параметр ndec задає кількість десяткових цифр у символьному записі числа. Якщо numb можна записати з точністю ndec цифр у формі числа з фіксованою крапкою, то число зображається у цій формі, інакше numb записується у формі числа з плаваючою крапкою. Точність ndec забезпечується шляхом округлення значення numb, кінцеві нулі дробової частини не виводяться. Повертає адресу першого байта сформованого рядка – buf.
getenv()	`char* getenv (const char* vname);` У разі успішного виконання повертає вказівник на початок внутрішнього символьного рядка, що зберігає значення змінної оточення, ім'я якої задає параметр vname. Якщо відповідну змінну не знайдено або зафіксовано помилку, то повертає NULL. Змінити значення або ім'я змінної оточення можна через функцію putenv().
itoa()	`char* itoa(int numb, char* str, int base);` Перетворює цілочислове значення параметра numb у символьний рядок і записує його за адресою, заданою параметром str. Параметр base задає основу системи числення, в якій буде записане число, він може приймати значення від 2 до 36. Від'ємні числа записуються зі знаком мінус тільки тоді, коли base дорівнює 10 (у разі інших основ числення двійкові коди від'ємних чисел розглядаються і перетворюються як беззнакові). Розмір str повинен бути достатнім для запису цифр числа й кінцевого '\0'. Повертає str. Контроль правильності аргументів не виконується, у разі помилкових значень результат не визначений.
labs()	`long int labs (long int number);` Повертає абсолютне значення параметра number, тип якого long int.

Функція	Прототип і призначення
`ldiv()`	`ldiv_t ldiv (long int numer, long int denom);` Аналог `div()`, але ділене, дільник, частка й залишок є довгими цілими числами (див. прим. 4).
`lfind()`	`void* lfind (const void* key, const void* arr, size_t* elnum, size_t size, int (*fcmp)(const void*, const void*));` Виконує лінійний (послідовний) пошук у невпорядкованому масиві даних, адресу початку якого задає параметр `arr`. Повертає адресу першого елемента `arr`, який збігається із ключем пошуку, заданим вказівником `key`, або `NULL`, якщо відповідного елемента в масиві немає. Параметр `elnum` є вказівником на кількість елементів масиву, а параметр `size` задає розмір елементів у байтах. Для порівняння елементів масиву із ключовим значенням `*key` використовується окрема функція, адресу якої задає останній параметр `fcmp`. Функція порівняння повинна повертати значення `<0`, якщо її перший аргумент менший за другий, `0`, якщо аргументи збігаються, і значення `>0`, якщо перший аргумент більший за другий.
`_lrotl()`	`unsigned long _lrotl (unsigned long numb, int count);` Циклічно зсуває двійкове значення беззнакового довгого цілого числа `numb` на `count` розрядів уліво (старші біти переносяться на місце зсунених молодших). Повертає значення, отримане в результаті зсування.
`_lrotr()`	`unsigned long _lrotr (unsigned long numb, int count);` Циклічно зсуває двійкове значення беззнакового довгого цілого числа `numb` на `count` розрядів управо (молодші біти переносяться на місце зсунених старших). Повертає значення, отримане в результаті зсування.
`lsearch()`	`void* lsearch (const void* key, const void* arr, size_t* elnum, size_t size, int (*fcmp)(const void*, const void*));` Аналог `lfind()`, виконує лінійний (послідовний) пошук ключового значення `*key` у невпорядкованому масиві даних `arr`. Якщо відповідний елемент відсутній, то він записується у кінець масиву. Повертає адресу першого елемента `arr`, який збігається із ключем пошуку.
`ltoa()`	`char* ltoa(long int numb, char* str, int base);` Аналог `itoa()`, але число `numb`, яке перетворюється у символьний рядок `str`, розглядається як довге ціле число.
`_makepath()`	`void _makepath (char* path, const char* drive, const char* dir, const char* name, const char* ext);` Формує повне ім'я файла зі заданих складових частин і заносить його у символьний рядок `path` у формі `D:\DIR\SUBDIR\FNAME.EXT`. Параметри функції: `drive` – рядок, що задає диск, може мати форму `D:` або `D` (`D` – літера диску, символ `:` долучається автоматично); `dir` – рядок, що містить імена каталогів і підкаталогів у формі `\DIR\SUBDIR\` (кінцевий `\` можна не записувати, він буде долучений автоматично); `name` – рядок, що задає ім'я файла (без розширення) `FNAME`; `ext` – рядок, що задає розширення файла `.EXT` або `EXT` (`.` долучається автоматично). Кожен із рядків: `drive`, `dir`, `name` чи `ext` може бути порожнім або рівним `NULL` – у цьому випадку відповідної складової у повному імені файла не буде.

Функція	Прототип і призначення
`malloc()`	`void* malloc (size_t size);` Виділяє у динамічній пам'яті неперервну ділянку, обсяг якої задає параметр `size`. Місце розташування динамічної ділянки визначається встановленою моделлю пам'яті (див. §14.5). У разі успішного виконання повертає адресу першого байта виділеної ділянки, а в разі невдачі – `NULL`.
`max()`	*тип_аргументів* `max(a, b);` Макрос, що визначає і повертає значення більшого з двох аргументів `a` і `b`.
`min()`	*тип_аргументів* `min(a, b);` Макрос, що визначає і повертає значення меншого з двох аргументів `a` і `b`.
`putenv()`	`int putenv(const char* env);` Долучає до зовнішнього оточення програми рядок `env` зі зміненим іменем чи значенням параметра середовища. Отримати змінене значення або ім'я змінної оточення можна через функцію `getenv()`.
`qsort()`	`void qsort (const void* arr, size_t nelem, size_t size, int (*fcmp)(const void*, const void*));` Виконує сортування елементів масиву `arr` за порядком зростання їх значень. Реалізує алгоритм швидкого сортування Ч.Хоара. Параметр `nelem` має задавати кількість елементів масиву, а параметр `size` – розмір елемента в байтах. Для порівняння елементів масиву в процесі сортування використовується окрема функція, адресу якої задає параметр `fcmp`. Функція порівняння повинна повертати значення <0, якщо її перший аргумент менший за другий, 0, якщо аргументи збігаються, і значення >0, якщо перший аргумент більший за другий. Якщо змінити знак значення функції порівняння на протилежний, то сортування буде виконуватися за порядком спадання значень елементів масиву `arr`.
`rand()`	`int rand(void);` Формує і повертає ціле псевдовипадкове число з діапазону `0..RAND_MAX` (див. прим. 5). Кожен наступний виклик функції `rand()` у процесі виконання програми приводить до генерування нового псевдовипадкового числа. Послідовно згенеровані числа рівномірно розподіляються у вказаному діапазоні.
`random()`	`int random (int rand_max);` Аналог `rand()`, але генерує і повертає псевдовипадкове число з діапазону `0..rand_max-1`, де `rand_max` задається як параметр функції.
`randomize()`	`void randomize(void);` Макрос, що ініціалізує генератор псевдовипадкових чисел випадковим значенням, сформованим на основі даних поточного часу, отриманих через функцію `time()`.
`realloc()`	`void* realloc (void* ptr, size_t newsize);` Змінює обсяг ділянки динамічної пам'яті, попередньо виділеної функцією `malloc()` або `calloc()`. Параметр `ptr` повинен вказувати на ділянку, обсяг якої треба змінити (збільшити або зменшити); параметр `newsize` задає новий обсяг ділянки в байтах. У разі успішного завершення функція повертає вказівник на ділянку нового обсягу, а в разі невдачі – `NULL`, стара ділянка динамічної пам'яті при цьому не змінюється (див. §13.1).

Функція	Прототип і призначення
`_rotl()`	`unsigned long _rotl (unsigned numb, int count);` Циклічно зсуває двійкове значення беззнакового цілого числа `numb` на `count` розрядів уліво (старші біти переносяться на місце зсунених молодших). Повертає значення, отримане в результаті зсування.
`_rotr()`	`unsigned long _rotr (unsigned numb, int count);` Циклічно зсуває двійкове значення беззнакового цілого числа `numb` на `count` розрядів управо (молодші біти переносяться на місце зсунених старших). Повертає значення, отримане в результаті зсування.
`_split-path()`	`void _splitpath (const char* path, char* drive, char* dir, char* name, char* ext);` Виділяє з повного імені файла, заданого символьним рядком `path`, складові частини: диск, шлях до файла, ім'я та розширення файла (повне ім'я повинно відповідати формі `D:\DIR\SUBDIR\FNAME.EXT`). Виділені компоненти заносяться як символьні рядки за адресами, заданими наступними параметрами функції: `drive` – ім'я диску у формі `D:`; `dir` – шлях до файла, що містить імена каталогів і підкаталогів у формі `\DIR\SUBDIR\`; `name` – ім'я файла у формі рядка `FNAME`; `ext` – розширення файла у формі `.EXT`. Якщо деякі компоненти в повному імені файла відсутні, то відповідні рядки будуть порожніми. Ділянки пам'яті, адреси яких задають параметри `drive`, `dir`, `name` та `ext`, повинні мати достатній розмір для запису відповідних символьних рядків.
`srand()`	`void srand (unsigned seed);` Встановлює стартове значення (його називають "затравкою") для функції генерування псевдовипадкових чисел `rand()` (стандартно `rand()` запускається зі значенням затравки 1). Значення `seed` визначає значення наступних псевдовипадкових чисел, згенерованих через виклик функції `rand()` або `random()`.
`strtod()`	`double strtod (const char* str, char** endp);` Виділяє зі символьного рядка `str` підрядок, що відповідає дійсному числу, і перетворює його у значення з типом `double`. Перед числом у `str` може бути записана довільна кількість символів роздільників. Саме число повинно бути записане як десяткове ціле чи дійсне у формі з фіксованою або з плаваючою крапкою. Кінцем числа вважається перший символ, який не належить до символів дійсного числа, адреса цього символа заноситься у вказівник, адресу якого задає параметр `endp`. У разі успішного перетворення функція повертає обчислене числове значення, а в разі виникнення переповнення – значення `±HUGE_VAL` і фіксує помилку переповнення. Якщо ж число виділити не вдалось, то функція повертає `0`.
`strtol()`	`long strtol (const char* str, char** endp, int base);` Виділяє зі символьного рядка `str` підрядок, що відповідає довгому цілому числу, і перетворює його у значення з типом `long`. Перед числом у `str` може бути записана довільна кількість символів роздільників. Параметр `base` визначає основу системи числення, в якій записано число, він може приймати значення від `2` до `36`. Цифрами числа можуть бути арабські цифри, а для `base>10` – послідовні латинські літери (малі або великі). Якщо `base` дорівнює `0`, то основа числа визначається формою його запису: число, що починається з `0`, вважається вісімковим, число з префіксом `0X` чи `0x` – шістнадцятковим, всі інші числа розглядаються як десяткові. Кінцем числа вважається перший

Функція	Прототип і призначення
	символ, який не належить до цифр даного числа. Адреса цього символа заноситься у змінну-вказівник, за адресою яку задає параметр `endp`. У разі успішного перетворення функція повертає обчислене числове значення, а в разі помилки – `0`.
`_strtold()`	`long double _strtold (const char* str, char** endp);` Аналог `strtod()`, але перетворює виділене зі символьного рядка `str` число у значення з типом `long double`.
`strtoul()`	`unsigned long strtoul (const char* str, char** endp, int base);` Аналог `strtol()`, але перетворює виділене зі символьного рядка `str` число у значення з типом `unsigned long`.
`swab()`	`void swab (char* fromadr, char* toadr, int nb);` Переписує `nb` байт (`nb` повинно бути парним числом) із ділянки пам'яті, заданої адресою `fromadr`, у ділянку, адресу якої задає параметр `toadr` (ділянки можуть перекриватись). У процесі переписування два сусідні байти (з парною і непарною адресами) міняються місцями.
`system()`	`int system (const char* command);` Виконує команду DOS, яка записана в символьному рядку `command`. У разі успішного виконання повертає значення `0`, а в разі невдачі – ненульове значення і фіксує у змінній `errno` код помилки.
`ultoa()`	`char* ultoa(unsigned long numb, char* str, int base);` Аналог `itoa()`, але число `numb`, яке перетворюється у символьний рядок `str`, розглядається як беззнакове довге ціле число.

Примітки:

1. `NULL` – макроконстанта, що позначає порожній вказівник. Її значення дорівнює `0`.
2. `size_t` – тип, що використовується для мобільності параметрів розміру даних і обсягу ділянок пам'яті. Задекларований через `typedef` як тип `unsigned int`.
3. `div_t` – тип значення функції `div()`, що задекларований так:
```
typedef struct {
            int quot;
            int rem;
        } div_t;
```
4. `ldiv_t` – тип значення функції `ldiv()`, що задекларований так:
```
typedef struct {
            long quot;
            long rem;
        } ldiv_t;
```
5. `RAND_MAX` – макроконстанта, що задає найбільше можливе випадкове число. Її значення – `0x7fff` дорівнює `INT_MAX`.

Таблиця Д2.5

Функції високорівневого потокоорієнтованого буферизованого введення/виведення даних (заголовний файл `<stdio.h>`)

Функція	Прототип і призначення
`clearerr()`	`void clearerr (FILE* fp);` Скидає в нуль значення внутрішніх індикаторів кінця файла і помилок введення/виведення, пов'язаних з потоком `fp` (ненульове значення відповідного індикатора сигналізує про кінець файла чи помилку обміну).
`fclose()`	`int fclose (FILE* fp);` Закриває потік `fp`. Попередньо скидає (дозаписує) внутрішні буфери цього потоку і закриває їх (буфери, що створені через `setbuf()` та `setvbuf()`, автоматично не скидаються). Закриває файл, пов'язаний з `fp`. У разі успішного виконання повертає `0`, а в разі невдачі (відсутній вільний дисковий простір, файл вже закрито або інше) – значення `EOF`.
`fcloseall()`	`int fcloseall(void);` Закриває усі відкриті програмою потоки, крім стандартних: `stdin`, `stdout`, `stderr`, `stdprn` та `stdaux` (див. прим. 2). У разі успішного завершення повертає кількість закритих файлів, а в разі помилки – `EOF`.
`feof()`	`int feof (FILE* fp);` Перевіряє, чи досягнуто кінця файла, пов'язаного з потоком `fp`. Повертає нуль, якщо не встановлено ознаку кінця файла, інакше – ненульове значення. Ознаку кінця файла знімають функції позиціювання: `fseek()`, `rewind()`, `fsetpos()` та функція скидання індикаторів `clearerr()`.
`ferror()`	`int ferror (FILE* fp);` Перевіряє, чи встановлено ознаку помилки в процесі звертання до потоку `fp`. Повертає нуль, якщо помилку не зафіксовано, та ненульове значення, якщо виявлено помилку. Щоб встановити причину помилки, треба звернутись до функції `perorr()`. Індикатор помилок скидають функції `rewind()` та `clearerr()`.
`fflush()`	`int fflush (FILE* fp);` Скидає вміст внутрішнього буфера, пов'язаного з потоком `fp`. Якщо `fp` відкрито для запису, то переносить із буфера виведення у файл цього потоку всі незаписані дані. Якщо ж `fp` є потоком введення, то очищає буфер потоку. Потоки залишаються відкритими. У разі успішного завершення повертає `0`, а в разі невдачі – значення `EOF`.
`fgetc()`	`int fgetc (FILE* fp);` Зчитує з потоку введення `fp` один символ і повертає його код у форматі `int`. У разі досягнення кінця файла або помилки читання повертає `EOF`.
`fgetchar()`	`int fgetchar(void);` Виконує виклик `fgetc(stdin)` – зчитує символ зі стандартного потоку введення `stdin` (найчастіше він пов'язаний з клавіатурою) і повертає код введеного символа в форматі `int`. У разі помилки введення повертає `EOF`.
`fgetpos()`	`int fgetpos (FILE* fp, fpos_t* fpos);` Записує за адресою, заданою вказівником `fpos`, значення покажчика поточної позиції файла, пов'язаного з потоком `fp`. Базовий тип вказівника `fpos` – `fpos_t` (див. прим. 4). Збережене значення можна використовувати надалі для відновлення зафіксованої позиції файла через виклик функції `fsetpos()`. Повертає `0` у разі успішного завершення і ненульове значення у разі виникнення помилки.

Функція	Прототип і призначення
`fgets()`	`char* fgets (char* buf, int max, FILE* fp);` Зчитує рядок символів з потоку введення `fp` і заносить його у буфер, адресу якого задає `buf` (розмір буфера повинен бути не меншим за `max`). Параметр `max` обмежує кількість символів, що може бути зчитана з потоку. Введення виконується до появи символа нового рядка чи символа кінця файла, але вводиться не більше, ніж `max-1` символів. Якщо зчитується символ `'\n'`, то він заноситься у `buf` і процес введення припиняється, інакше послідовно зчитуються `max-1` символів потоку. У кінець введеного рядка заноситься `'\0'`. У разі успішного виконання `fgets()` повертає `buf` – адресу початку введеного рядка, а в разі помилки, зокрема, якщо досягнуто кінця файла – значення `NULL`.
`fileno()`	`int fileno (FILE* fp);` Повертає дескриптор файла, пов'язаного з потоком `fp` (номер, за яким цей файл зареєстровано у внутрішній таблиці відкритих файлів DOS). Дескриптор дає змогу застосовувати до файла необхідні функції низькорівневої роботи з файлами, оголошені в `<io.h>`.
`flushall()`	`int flushall(void);` Скидає вміст буферів усіх відкритих файлів. Буфери потоків введення очищаються, а буфери потоків виведення дозаписуються у відповідні файли. Потоки залишаються відкритими. Повертає кількість відкритих потоків.
`fopen()`	`FILE* fopen (const char* fname, const char* fmode);` Відкриває для обміну даними файл, ім'я якого задає `fname` (`fname` може включати шлях до файла), і створює потік, пов'язаний з цим файлом. Параметр `fmode` встановлює режим відкриття потоку і визначає операції, які можна виконувати з даним потоком. Якщо рядок `fmode` починається літерою `r`, то файл з іменем `fname` повинен вже існувати, оскільки потік відкривається для читання; якщо `fmode` починається літерою `w`, то файл відкривається для запису: вміст існуючого файла витирається або створюється новий файл з іменем `fname`; якщо ж режим відкриття файла починається літерою `a`, то файл відкривається для доповнення: вміст існуючого файла зберігається або створюється новий файл. Можна відкрити файл відразу для читання і запису, вказавши в рядку `fmode` символ `+`, а також встановити текстовий чи бінарний режим обміну даними з файлом (детальніша інформація у §15.3). Граничну кількість файлів, що можуть бути відкриті одночасно, задає макроконстанта `FOPEN_MAX`. У разі успішного виконання `fopen()` повертає вказівник на структуру `FILE`, в яку заноситься інформація про відкритий файл і створений потік. Якщо ж файл відкрити не вдалось, то повертає `NULL`.
`fprintf()`	`int fprintf (FILE* fp, const char* format, ...);` Аналог `printf()`. Виконує форматне виведення даних у файл, пов'язаний з потоком виведення `fp`. Параметр `format` вказує на текстовий рядок, в якому записані специфікації форматних перетворень. За рядком формату вказуються вирази, значення яких мають бути виведені в потік у формі, що задається вмістом рядка `format`. Ці параметри не є обов'язковими, їх кількість визначається специфікаціями рядка формату. Правила формування рядка `format`, структуру специфікацій виведення, вимоги щодо їх взаємоузгодження з типами даних, що виводяться, та інші властивості форматних перетворень даних детально описано в §5.1. Функція `fprintf()` повертає кількість символів, виведених у потік `fp`, або `EOF`, якщо зафіксовано помилку виведення.

Функція	Прототип і призначення
`fputc()`	`int fputc (int sym, FILE * fp);` Записує символ, код якого задає молодший байт параметра `sym` у файл, пов'язаний з потоком `fp`. Повертає код записаного символа (`sym`) у разі успішного виконання та значення `EOF`, якщо вивести символ не вдалось.
`fputchar()`	`int fputchar (int sym);` Виконує виклик `fputc(sym, stdout)` – записує символ `sym`, у стандартний потік виведення `stdout` (найчастіше це екран). Повертає код записаного символа в разі успішного виконання та `EOF`, якщо зафіксовано помилку виведення.
`fputs()`	`int fputs (const char * str, FILE * fp);` Записує символьний рядок, адресу якого задає параметр `str`, у файл, пов'язаний з потоком виведення `fp`. Рядок повинен завершуватись нуль-символом, який не записується у файл і не перетворюється в інші символи. У разі успішного виконання повертає код останнього записаного символа, а в разі помилки виведення – значення `EOF`.
`fread()`	`size_t fread (void * buf, size_t size, size_t n, FILE * fp);` Зчитує з файла, пов'язаного з потоком `fp`, `n` об'єктів, кожен з яких має розмір `size` байт (див. прим. 6), і записує їх у буфер, адресу якого задає вказівник `buf`. Повертає кількість реально введених об'єктів. Якщо результат менший за `n`, то це означає, що досягнуто кінця файла або зафіксовано помилку введення.
`freopen()`	`FILE * freopen (const char * fname, const char * fmode, FILE * fp);` Виконує перескерування потоку – пов'язує відкритий потік `fp` з іншим файлом, ім'я якого задає параметр `fname`. Якщо це потрібно, то закриває файл, з яким попередньо був пов'язаний потік `fp`. Параметр `fmode` встановлює режим відкриття нового файла, він є повністю ідентичним до відповідного параметра функції `fopen()`. Найчастіше виконують перескерування стандартних потоків `stdin`, `stdout` та `stderr` у певний заданий файл. У разі успішного виконання `freopen()` повертає вказівник на потік, пов'язаний з новим файлом, а у випадку помилки – значення `NULL`.
`fscanf()`	`int fscanf (FILE * fp, const char * format, ...);` Аналог `scanf()`. Виконує введення і форматне перетворення даних із текстового файла, пов'язаного з потоком `fp`. Параметр `format` вказує на текстовий рядок, в якому записано послідовність специфікацій форматних перетворень. Наступні параметри у виклику функції задають адреси змінних або ділянок пам'яті, куди будуть записуватись введені значення. Ці параметри не є обов'язковими, їх кількість визначається специфікаціями рядка `format`. Для кожного поля введення потоку `fp` у рядку `format` повинна бути вказана відповідна за типом специфікація, а в списку введення – адреса запису даного (якщо це поле не пропускається). Структура специфікацій формату, правила форматних перетворень введених даних та їх взаємоузгодження зі списком введення детально описані в §5.2. Функція `fscanf()` повертає кількість полів введення, які були прочитані, перетворені й записані у відповідні змінні (пропущені поля не враховуються); якщо не записано ні одного даного, то повертає `0`; якщо ж перед читанням першого поля зафіксовано кінець файла, то повертає значення `EOF`.

Функція	Прототип і призначення
`fseek()`	`int fseek (FILE* fp, long offset, int base);` Змінює значення покажчика поточної позиції файла, пов'язаного з потоком `fp`. Параметр `offset` задає кількість байтів, на які треба змістити поточну позицію файла: якщо `offset > 0`, то покажчик поточної позиції зсувається у напрямку кінця файла, якщо `offset < 0`, то покажчик зсувається до початку файла. Параметр `base` задає базис, відносно якого здійснюється зсунення покажчика, він може приймати одне з трьох значень: `SEEK_SET` – зміщення відносно початку файла, `SEEK_CUR` – зміщення відносно поточної позиції файла, `SEEK_END` – зміщення відносно кінця файла (див. прим. 7). Наступні операції обміну даними з файлом будуть виконуватись, починаючи зі встановленої позиції. Повертає `0` у разі успішного завершення і ненульове значення у разі виникнення помилки позиціювання.
`fsetpos()`	`int fsetpos (FILE* fp, const fpos_t* fpos);` Встановлює покажчик поточної позиції файла, пов'язаного з потоком `fp`, на позицію, зафіксовану раніше через виклик функції `fgetpos()`. Вказівник `fpos` задає адресу, за якою записано значення зафіксованої позиції файла. Наступні операції обміну даними з файлом будуть виконуватись, починаючи зі встановленої позиції. Повертає `0` у разі успішного завершення і ненульове значення у разі виникнення помилки позиціювання.
`ftell()`	`long ftell (FILE* fp);` Повертає значення покажчика поточної позиції файла, пов'язаного з потоком `fp`. Якщо потік відкрито в бінарному режимі, то значення `ftell()` дорівнює кількості байтів від початку файла до його поточної позиції. Для текстових потоків можуть бути відхилення, спричинені перетворенням символів кінця рядка. Значення `ftell()` можна використовувати для позиціювання покажчика файла функцією `fseek()`. Якщо ж зафіксовано помилку виконання, то функція повертає значення `-1L`.
`fwrite()`	`size_t fwrite (void* buf, size_t size, size_t n, FILE* fp);` Переписує з буфера, адресу якого задає вказівник `buf`, у файл, пов'язаний з потоком `fp`, `n` об'єктів, кожен з яких має розмір `size` байт. Повертає кількість реально записаних у файл об'єктів. Якщо це значення менше за `n`, то досягнуто кінця файла або зафіксовано помилку обміну даними.
`getc()`	`int getc (FILE* fp);` Зчитує з потоку введення `fp` один символ і повертає його код. У разі досягнення кінця файла або фіксації помилки читання повертає `EOF`. Реалізовано як макрос, що виконує виклик `fgetc(fp)`.
`getchar()`	`int getchar (void);` Зчитує один символ зі стандартного потоку введення `stdin` (найчастіше це клавіатура) і повертає його код. У разі помилки введення повертає `EOF`. Реалізовано як макрос, що виконує виклик `fgetc(stdin)`.
`gets()`	`char* gets (char* str);` Зчитує зі стандартного потоку введення `stdin` (найчастіше він пов'язаний з клавіатурою) послідовність символів до появи символа нового рядка (символа клавіші *Enter*) і записує введений рядок за адресою, заданою параметром `str`. Символ нового рядка (`\n`) замінюється кінцевим нуль-символом (`\0`). Повертає адресу початку введеного рядка (`str`) або `NULL`, якщо зафіксовано помилку.

Функція	Прототип і призначення
`getw()`	`int getw (FILE* fp);` Зчитує з файла, пов'язаного з потоком `fp` і відкритого в бінарному режимі, послідовність байтів, що відповідає двійковому коду даного з типом `int`. Повертає введене ціле число або `EOF`, якщо зафіксовано помилку.
`perror()`	`void perror (const char* errtext);` Виводить у стандартний потік помилок `stderr` (найчастіше він пов'язаний з екраном) текст повідомлення, записаний в `errtext`, а за ним після двокрапки стандартне системне роз'яснення причини виникнення помилки.
`printf()`	`int printf (const char* format, ...);` Виконує форматне виведення даних у стандартний вихідний потік `stdout` (здебільшого він пов'язаний з виведенням на екран). Параметр `format` вказує на текстовий рядок, який може включати три групи символів: звичайні ASCII-символи – вони виводяться без змін; керуючі символи – виконують відповідну дію; специфікації формату – задають інтерпретацію і форму зображення відповідного параметра зі списку виведення. За рядком формату записуються вирази, значення яких мають бути виведені в потік. Ці параметри є необов'язковими, їх кількість визначається специфікаціями рядка `format` – кожній специфікації формату повинен відповідати один сумісний за типом вираз зі списку виведення. Особливості формування рядка `format`, структуру специфікацій виведення, правила їх взаємоузгодження з даними, що виводяться, та інші властивості форматних перетворень детально описано в §5.1. Функція повертає кількість реально виведених символів або `EOF`, якщо зафіксовано помилку виведення.
`putc()`	`int putc (int sym, FILE* fp);` Записує символ `sym` у файл, пов'язаний з потоком виведення `fp`. Повертає код записаного символа (`sym`) або `EOF`, якщо вивести символ не вдалось. Реалізовано як макрос, що виконує виклик `fputc(sym, fp)`.
`putchar()`	`int putchar (int sym);` Записує символ, код якого задає молодший байт параметра `sym`, у стандартний потік виведення `stdout` (найчастіше це екран). Повертає код записаного символа (`sym`) або `EOF`, якщо вивести символ не вдалось. Реалізовано як макрос, що виконує виклик `fputc(sym, stdout)`.
`puts()`	`int puts (const char* str);` Виводить символьний рядок, адресу якого задає `str`, у стандартний потік виведення `stdout` (найчастіше це екран). Рядок повинен обов'язково завершуватись нуль-символом, який перетворюється у символ нового рядка. Повертає додатне значення у разі успішного виконання та `EOF`, якщо зафіксовано помилку.
`putw()`	`int putw (int num, FILE* fp);` Записує двійковий код заданого цілого числа `num` у бінарний файл, пов'язаний з потоком виведення `fp`. Повертає виведене значення або `EOF`, якщо зафіксовано помилку.
`remove()`	`int remove (const char* fname);` Витирає файл, ім'я якого задає параметр `fname`. Файл повинен бути закритим. Реалізовано як макрос, що викликає функцію `unlink(fname)`.
`rename()`	`int rename (const char* oldname, const char* newname);` Перейменовує файл, де `oldname` – старе ім'я файла, а `newname` – нове ім'я. Файл повинен бути закритим. Кожне з імен може включати повний чи

Функція	Прототип і призначення
	скорочений шлях до файла. Нове ім'я файла `newname` має бути унікальним для каталога, в який записується файл.
`rewind()`	`void rewind(FILE* fp);` Встановлює покажчик поточної позиції файла, пов'язаного з потоком `fp`, на початок даного файла і скидає в нуль значення внутрішніх індикаторів кінця файла і помилок введення/виведення потоку `fp`.
`rmtmp()`	`int rmtmp(void);` Закриває і витирає всі тимчасові потоки та файли, створені в процесі виконання програми через звертання до функції `tmpfile()`. Повертає кількість витертих тимчасових файлів.
`scanf()`	`int scanf (const char* format, ...);` Виконує введення і форматне перетворення даних зі стандартного вхідного потоку `stdin` (переважно він пов'язаний з клавіатурою). Послідовно зчитує введені символи, формує з них поля введення і перетворює їх у двійкову форму відповідно до специфікацій, заданих у рядку формату `format`. За умови успішного перетворення записує отримані значення у змінні або ділянки пам'яті, адреси яких задаються у списку введення (список введення може складатись зі змінної кількості аргументів). Забезпечує введення даних усіх числових типів, а також одиночних символів і символьних рядків (без символа пробілу). Параметр `format` вказує на символьний рядок, що складається зі специфікацій, які визначають спосіб інтерпретації введених даних. Наступні параметри у звертанні до функції задають адреси, за якими будуть записуватись введені значення. Для кожного поля введення у рядку `format` повинна бути вказана відповідна за типом специфікація, а в списку введення – адреса запису даного (якщо це поле не пропускається). Структура специфікацій формату, правила форматних перетворень введених даних та їх взаємоузгодження зі списком введення детально описані в §5.2. Функція `scanf()` повертає кількість полів введення, які були успішно прочитані, перетворені й записані у відповідні змінні. Якщо не записано ні одного даного, то повертає `0`, а якщо перед першим полем введення зчитано символ кінця файла, то – `EOF`.
`setbuf()`	`void setbuf (FILE* fp, char* pbuf);` Встановлює для відкритого потоку `fp` користувацький буфер, адресу якого задає параметр `pbuf`. Розмір користувацького буфера повинен бути не меншим за `BUFSIZ` (див. прим. 8). Якщо `pbuf` має значення `NULL`, то потік буде небуферизованим.
`setvbuf()`	`int setvbuf (FILE* fp, char* pbuf, int bmode, size_t bsize);` Керує буферизацією даних для відкритого потоку `fp` і дає змогу встановити буфер, визначений користувачем. Параметр `pbuf` задає адресу початку буфера, а `bsize` – розмір буфера. Якщо значення `pbuf` дорівнює `NULL`, то автоматично відкриває в динамічній пам'яті буфер заданого розміру. Параметр `bmode` встановлює режим буферизації, він може приймати такі значення: `_IOFBF` – звичайна буферизація, `_IOLBF` – рядкова буферизація, `_IONBF` – буферизація не виконується (див. прим. 9). Повертає `0` у разі успішного завершення і ненульове значення, якщо параметри функції задано неправильно.
`sprintf()`	`int sprintf (char* str, const char* format, ...);` Аналог функції `printf()`, але дані виводяться не в стандартний потік `stdout`, а записуються у символьний рядок, адресу якого задає параметр `str` (треба забезпечити, щоб розмір `str` був достатнім для запису всього рядка виведення). У кінець сформованого рядка записує `'\0'`.

Функція	Прототип і призначення
`sscanf()`	`int sscanf (char* str, const char* format, ...);` Аналог функції `scanf()`, але дані вводяться не зі стандартного потоку `stdin`, а зчитуються зі символьного рядка `str`. Рядок даних має бути заповнений згідно зі специфікаціями, вказаними в рядку `format` (див. §5.2). Повертає кількість полів введення з рядка `str`, які були прочитані, перетворені й записані у відповідні змінні (пропущені поля не враховуються), або `EOF`, якщо перед читанням першого поля зафіксовано кінець рядка.
`strerror()`	`char* strerror (int errnum);` Повертає вказівник на внутрішній статичний рядок, в якому записано стандартне повідомлення про помилку, номер якої задає параметр `errnum`.
`_strerror()`	`char* _strerror (const char* errtext);` Формує повідомлення про останню зафіксовану системою помилку роботи з потоками і записує його у внутрішній статичний рядок, який оновлюється з кожним викликом функції. Рядок повідомлення буде складатись із тексту, записаного в `errtext` (його довжина не повинна перевищувати 94 символи), за яким після двокрапки вказується стандартне системне роз'яснення причини помилки. Повертає адресу сформованого рядка.
`tempnam()`	`char* tempnam (char* dir, char* prefix);` Генерує ім'я файла, унікальне для каталога, заданого параметром `dir`. Ім'я файла буде починатись префіксом, який задає рядок `prefix` (він може складатись із 0..5 символів). До префікса буде долучено ще 6 символів. У разі успішного виконання повертає вказівник на рядок у динамічній пам'яті, де записано згенероване ім'я, інакше – `NULL`.
`tmpfile()`	`FILE* tmpfile(void);` Створює тимчасовий файл для запису й читання та відкриває для нього потік у режимі `"w+b"`. Файл отримує унікальне ім'я. Кількість одночасно відкритих тимчасових файлів не може перевищувати `FOPEN_MAX`. Створений файл автоматично витирається після закривання та під час завершення роботи програми. У разі успішного виконання повертає вказівник на потік, пов'язаний із створеним файлом, а в разі помилки – `NULL`.
`tmpnam()`	`char* tmpnam (char* fname);` Генерує унікальне ім'я файла та записує його за адресою, заданою параметром `fname`. Розмір масиву `fname` повинен бути не меншим за `L_tmpnam` символів. Якщо значення `fname` дорівнює `NULL`, то згенероване ім'я записується у внутрішню статичну змінну, значення якої оновлюється з кожним викликом `tmpnam()`. Можна згенерувати до `TMP_MAX` (див. прим. 12) унікальних імен. Повертає адресу початку рядка, в який записано згенероване ім'я, або `NULL` у разі невдачі.
`ungetc()`	`int ungetc (int sym, FILE* fp);` Передає (повертає) у потік введення, пов'язаний з `fp`, символ, код якого задається молодшим байтом параметра `sym`. Цей символ буде зчитаний першим довільною наступною операцією введення з потоку `fp`. Можна передати тільки один символ (якщо не виконувалось зчитування даних, то наступний переданий символ затре попередній). У разі успішного виконання функція повертає `sym`, а в разі помилки – значення `EOF`.
`unlink()`	`int unlink (const char* fname);` Витирає файл, ім'я якого задає параметр `fname`. Файл обов'язково має бути закритим. Повертає `0` у разі успішного виконання та `-1`, якщо файл витерти не вдалось.

Функція	Прототип і призначення

vfprintf()
vprintf()
vsprintf()
	`int vfprintf(FILE* fp, const* char* format, va_list vaptr);`
	`int vprintf (const char* format, va_list vaptr);`
	`int vsprintf (char* str, const char* format, va_list vaptr);`
	Призначення цих функцій таке саме, як і відповідних функцій форматного виведення: `fprintf()`, `printf()` та `sprintf()`. Різниця полягає тільки в тому, що останній параметр функцій `vfprintf()`, `vprintf()` та `vsprintf()` – `vaptr` має тип `va_list`, визначений у заголовному файлі `<stdarg.h>`, і вказує на початок списку неоголошених аргументів (див. §11.11.2). Наведені функції дають змогу реалізовувати форматне виведення даних у користувацьких функціях зі змінною кількістю параметрів.

vfscanf()
vscanf()
vsscanf()
	`int vfscanf(FILE* fp, const * char* format, va_list vaptr);`
	`int vscanf (const char* format, va_list vaptr);`
	`int vsscanf (char* str, const char* format, va_list vaptr);`
	Призначення цих функцій таке саме, як і відповідних функцій форматного введення даних: `fscanf()`, `scanf()` та `sscanf()`. Різниця полягає тільки в тому, що останній параметр функцій `vfscanf()`, `vscanf()` та `vsscanf()` – `vaptr` має тип `va_list`, визначений у заголовному файлі `<stdarg.h>`, і вказує на початок списку неоголошених аргументів (див. §11.11.2). Наведені функції дають змогу реалізовувати форматне виведення даних у користувацьких функціях зі змінною кількістю параметрів.

Примітки:

1. `FILE` – структура, призначена для збереження інформації про відкритий файл і відповідний потік. Її задекларовано так:

```
typedef struct {
    int level;                          /* індикатор заповнення буфера */
    unsigned flags;                     /* слово прапорців стану потоку:
                                режими відкриття й обміну, спосіб буферизації,
                                    індикатори кінця файла та помилок тощо */
    char fd;                                    /* дескриптор файла */
    unsigned char hold;                         /* повернений символ */
    int bsize;                                  /* ємність буфера */
    unsigned char far* buffer;                  /* вказівник на буфер */
    unsigned char far* curp;    /* покажчик поточної позиції обміну */
    unsigned istemp;                    /* індикатор тимчасового файла */
    short token;                        /* індикатор дійсності файла */
} FILE;
```

2. `stdin`, `stdout`, `stderr`, `stdprn` та `stdaux` – стандартні потоки: введення, виведення, повідомлень про помилки, друкування та додаткового пристрою введення/виведення, які автоматично відкриваються на початку виконання кожної програми (див. §15.4).
3. `EOF` – макроконстанта, що використовується у багатьох функціях введення/виведення даних як індикатор помилок, зокрема досягнення кінця файла. Її значення дорівнює `-1`.

4. `fpos_t` – тип, що використовується для мобільності параметра поточної позиції файла. Задекларований через `typedef` як тип `long int`.
5. `NULL` – макроконстанта, що позначає порожній вказівник. Її значення дорівнює `0`.
6. `size_t` – тип, що використовується для мобільності параметрів розміру даних. Задекларований через `typedef` як тип `unsigned int`.
7. `SEEK_SET`, `SEEK_CUR`, `SEEK_END` – макроконстанти, що задають базис, відносно якого функція `fseek()` здійснює зміщення покажчика поточної позиції файла. Значення цих макроконстант дорівнюють відповідно `0`, `1` та `2`.
8. `BUFSIZ` – макроконстанта, що визначає мінімальну ємність буфера потоку для функції `setbuf()`. Її значення дорівнює `512`.
9. `_IOFBF`, `_IOLBF`, `_IONBF` – макроконстанти режимів буферизації (див. §15.8). Значення цих макроконстант дорівнюють відповідно `0`, `1` та `2`.
10. `FOPEN_MAX` – макроконстанта, що визначає максимальну кількість файлів, які можуть бути одночасно відкриті програмою. Для Borland C це значення не перевищує `20`.
11. `L_tmpnam` – макроконстанта, що визначає розмір символьного рядка, достатній для збереження унікального імені файла, згенерованого функцією `tmpnam()`. Її значення дорівнює `13`.
12. `TMP_MAX` – макроконстанта, що визначає максимальну кількість унікальних імен файлів, які можна згенерувати функцією `tmpnam()`. Її значення дорівнює `0xffff`.

Таблиця Д2.6

Функції дати та часу
(заголовний файл `<time.h>`)

Функція	Прототип і призначення
`asctime()`	`char* asctime (const struct tm* tmdata);` Перетворює параметри календарного часу й дати, записані в структурі, на яку вказує `tmdata` (див. прим. 1), у 26-символьний рядок і записує його у спеціальну внутрішню статичну змінну, вміст якої оновлюється з кожним викликом функції. Форма рядка результату наступна: `DDD MMM dd hh:mm:ss YYYY`, де `DDD` – найменування дня тижня (`Mon`, `Tue`, `Wed` і т.д.), `MMM` – місяць (`Jan`, `Feb`, `Mar` і т.д.), `hh` – година (`0..23`), `mm` – хвилини (`0..59`), `ss` – секунди (`0..59`), `YYYY` – рік. У кінець рядка записується пара символів `n\0`. Повертає адресу сформованого рядка.
`clock()`	`clock_t clock(void);` Повертає час процесора в т. зв. "тіках" (див. прим. 3), що минув від початку запуску програми. Щоб перевести це значення в секунди, треба поділити його на макроконстанту `CLK_TCK` (або `CLOCKS_PER_SEK`). Якщо час визначити неможливо, то повертає `-1`.
`ctime()`	`char* ctime (const time_t* ptime);` Перетворює календарний час, що зберігається за адресою, яку задає вказівник `ptime` (див. прим. 2), у 26-символьний рядок і записує його у спеціальну внутрішню статичну змінну, вміст якої оновлюється з кожним викликом функції. Повертає адресу сформованого рядка. Вміст рядка результату наступний: `DDD MMM dd hh:mm:ss YYYY`, де `DDD` – найменування дня тижня (`Mon`, `Tue`, `Wed` і т.д.), `MMM` – місяць (`Jan`, `Feb`, `Mar` і т.д.), `hh` – година (`0..23`), `mm` – хвилини (`0..59`), `ss` – секунди (`0..59`), `YYYY` – рік. У кінець рядка записується пара символів `\n\0`.

Функція	Прототип і призначення
`difftime()`	`double difftime (time_t time2, time_t time1);` Обчислює і повертає у формі дійсного числа значення у секундах різниці параметрів `time2 - time1`. Обидва параметри повинні мати тип `time_t` (див. прим. 2).
`gmtime()`	`struct tm* gmtime (const time_t* ptime);` Формує внутрішню статичну структуру (див. прим. 1), у поля якої заносить календарні й часові дані у формі часу за Грінвічем (GMT – Greenwich Mean Time), що відповідають поточному значенню змінної, адресу якої задає параметр `ptime` (див. прим. 2). Здебільшого як значення `*ptime` використовують значення, повернене функцією `time()`. `gmtime()` повертає адресу сформованої структури або `NULL`, якщо виявлено помилку. Вміст внутрішньої структури оновлюється з кожним викликом функції.
`localtime()`	`struct tm* localtime (const time_t* ptime);` Аналог `gmtime()`, але заповнює вихідну структуру даними, що відповідають локальному (місцевому) часу.
`mktime()`	`time_t mktime (struct tm* tmdata);` Повертає значення календарного часу в форматі `time_t` (див. прим. 2). Результат формується зі значень полів структури, адресу якої задає параметр `tmdata`. Якщо значення певних полів структури `*tmdata` не відповідають встановленим діапазонам, то виконується їх перетворення (наприклад, 123 сек. перетворюються у 2 хв. і 3 сек.), поля `tm_wday` і `tm_yday` (див. прим. 1) заповнюються автоматично на основі інших даних. Якщо ж перетворення виконати неможливо, то функція повертає `-1`.
`strftime()`	`size_t strftime (char* sttm, size_t maxsyn, const char* fmt, const struct tm* tmdata);` Формує з даних структури, на яку вказує `tmdata` (див. прим. 1), символьний рядок з інформацією про час і дату та записує його за адресою, заданою параметром `sttm`. Параметр `maxsym` обмежує кількість символів, які будуть записані в `sttm`. Параметр `fmt` вказує на рядок формату, який (подібно до рядка формату функції `sprintf()`) визначає форму запису даних у `sttm` та їх форматні перетворення. Повертає кількість символів, записаних у `sttm`, або `0` у разі помилки.
`_strdate()`	`char* _strdate (char* buf);` Записує у буфер `buf` (розмір буфера повинен бути не меншим за 9 символів) поточну дату у формі `MM/DD/YY`, де `MM` – номер місяця, `DD` – день місяця, `YY` – рік, усі три значення двоцифрові. Повертає `buf` – адресу початку сформованого рядка.
`_strtime()`	`char* _strtime (char* buf);` Записує у буфер `buf` (розмір буфера повинен бути не меншим за 9 символів) поточний час у формі `HH:MM:SS`, де `HH` – години, `MM` – хвилини, `SS` – секунди, усі три значення двоцифрові. Повертає `buf` – адресу початку сформованого рядка.
`stime()`	`int stime (time_t* pnewtm);` Встановлює поточний системний час. Нове значення часу зчитується за адресою, яку задає параметр `pnewtm`, що має базовий тип `time_t` (див. прим. 2). Повертає `0` у разі успішного виконання.

Функція	Прототип і призначення
`time()`	`time_t time (time_t* ptime);` Повертає поточний системний час як значення з типом `time_t` – кількість секунд, що минули від 1 січня 1970 року за Грінвічем. Якщо параметр `ptime` не дорівнює `NULL`, то записує значення результату в змінну, задану адресою `ptime`.

Примітки:

1. `struct tm` – структура, призначена для збереження часових і календарних параметрів (т. зв. розподілений час). Склад її полів такий:

```
struct tm {
   int tm_sec;      /* секунди (0..59) */
   int tm_min;      /* хвилини (0..59) */
   int tm_hour;     /* години (0..23) */
   int tm_mday;     /* день місяця (1..31) */
   int tm_mon;      /* місяць (0..11) */
   int tm_year;     /* рік (календарний рік-1900) */
   int tm_wday;     /* день тижня (0..6, де 0 - неділя) */
   int tm_yday;     /* день року (0..365) */
   int tm_isdst;    /* 0, якщо не застосовується літній час */
};
```

2. `time_t` – тип значення функції `time()`, задає т. зв. календарний час. Задекларований через `typedef` як тип `long int`.
3. `clock_t` – тип значення функції `clock()`, задає приблизний час виконання програми в т. зв. "тіках". Задекларований через `typedef` як тип `long int`.
4. `CLK_TCK` та `CLOCKS_PER_SEK` – макроконстанти, які задають кількість відліків ("тіків") системного таймера за 1 секунду. Їх значення дорівнюють `18.2`.
5. `size_t` – тип, що використовується для мобільності параметрів розміру даних. Задекларований через `typedef` як тип `unsigned int`.

Таблиця Д2.7

Функції консольного введення/виведення даних (заголовний файл `<conio.h>`)

Функція	Прототип і призначення
`cgets()`	`char* cgets (char* buf);` Записує введений з клавіатури символьний рядок і його довжину у ділянку оперативної пам'яті, адресу початку якої задає вказівник `buf`. Перед викликом функції у `buf[0]` треба занести розмір ділянки введення – це значення обмежуватиме довжину введеного рядка. Символи рядка заносяться в буфер, починаючи з елемента `buf[2]`, можна ввести не більше, ніж `buf[0]-1` символів. У кінець введеного рядка автоматично записується `'\0'`. Введення завершується, коли натиснено клавішу *Enter*. Текстовий курсор залишається за останнім відтвореним на екрані символом. У `buf[1]` записується реальна довжина введеного рядка (інші властивості `cgets()` див. у §16.2.2). За умови успішного виконання функція повертає вказівник на перший символ введеного рядка (адресу `buf+2`), а в противному разі – `NULL`.

Функція	Прототип і призначення
`clreol()`	`void clreol(void);` Витирає кінцеву частину рядка активного текстового вікна, починаючи від поточної позиції текстового курсора. Курсор зберігає свою позицію.
`clrscr()`	`void clrscr(void);` Очищає активне текстове вікно та зафарбовує його поточним кольором фону. Курсор встановлюється в позицію вікна (1, 1).
`cprintf()`	`int cprintf(const char* format, ...);` Виконує форматне виведення даних у межах активного текстового вікна. Параметр `format` вказує на текстовий рядок, в якому записані специфікації форматних перетворень. За рядком формату вказуються вирази, значення яких мають бути перетворені та відображені у формі, що задається вмістом рядка `format`. Аналог функції `printf()` з `<stdio.h>` (див. табл. Д2.5), але додатково реалізує можливості консольного виведення даних (див. §16.1.3). Для переведення курсора на початок нового рядка треба використовувати комбінацію символів `"\n\r"` або `"\r\n"`. Повертає загальну кількість виведених символів або `EOF`, якщо зафіксовано помилку виведення.
`cputs()`	`int cputs(const char* str);` Виводить символьний рядок, адресу початку якого задає вказівник `str`, у активне вікно екрана. Для відображення рядка використовуються поточні кольори символів та фону. Після виведення залишає текстовий курсор за останнім відтвореним символом. Для встановлення курсора на початок нового рядка треба вивести пару символів `"\n\r"` або `"\r\n"`. Повертає код останнього виведеного у вікно символа.
`cscanf()`	`int cscanf(const char* format, ...);` Виконує форматне введення даних з клавіатури та відображає зчитані символи в активному текстовому вікні. Параметр `format` вказує на текстовий рядок, в якому записано послідовність специфікацій форматних перетворень. Наступні параметри у звертанні до функції задають адреси змінних або ділянок пам'яті, куди будуть записуватись введені значення. Форматні перетворення даних `cscanf()` виконує так само, як функція `scanf()` (див. табл. Д2.5), але не здійснює буферизації і опрацьовує кожен символ відразу після натискання на клавішу (див. §16.2.2). Повертає кількість даних, які були успішно прочитані, перетворені й записані у відповідні змінні. Якщо не записано ні одного даного, то повертає 0, а якщо перед першим полем введення зчитано символ кінця файла, то – `EOF`.
`delline()`	`void delline(void);` Витирає рядок активного вікна, на якому в даний момент розміщений текстовий курсор. Всі наступні рядки цього вікна зсуваються на один угору. Курсор встановлюється на початок рядка, який пересунувся на місце видаленого.
`getch()`	`int getch(void);` Зчитує і повертає код одного символа з буфера клавіатури без відображення його на екрані. Якщо буфер клавіатури порожній, то призупиняє виконання програми і переходить у стан очікування натиснення клавіші. Незалежно від натисненої клавіші позиція текстового курсора на екрані не змінюється. Функція повертає ASCII-код введеного символа, якщо натиснуто символьну клавішу, або нуль, якщо натиснуто одну з керуючих або функціональних клавіш чи клавішних комбінацій з набору розширеної клавіатури (див. табл. 16.3). В цьому випадку потрібно ще раз викликати `getch()`, щоб отримати розширений ASCII-код клавіші (див. §16.2.2).

Функція	Прототип і призначення
`getche()`	`int getche(void);` Аналог `getch()`, але відображає на екрані введений з клавіатури символ.
`getpass()`	`char* getpass (const char* prompt);` Записує введений з клавіатури пароль у внутрішній статичний рядок. Попередньо виводить на екран текст підказки `prompt`. Символи пароля не відображаються, можна ввести не більше, ніж 8 символів. Повертає адресу першого символа введеного пароля.
`gettext()`	`int gettext (int left, int top, int right, int bot, void* buf);` Послідовно записує в буфер, адресу якого задає вказівник `buf`, символи й атрибути прямокутної області екрана, заданої координатами лівого верхнього (`left, top`) та правого нижнього (`right, bot`) кутів. Координати області копіювання повноекранні. Буфер, на який вказує `buf`, повинен бути попередньо виділений у статичній або динамічній пам'яті. Необхідний мінімальний розмір буфера становить `(right-left+1)` × × `(bot-top+1)` × 2 байт. У разі успішного копіювання заданого вікна повертає 1, а в разі помилки копіювання (зокрема, якщо екранні координати задано неправильно) повертає 0.
`gettext-info()`	`void gettextinfo (struct text_info* winf);` Записує в структуру зі шаблоном `struct text_info` (див. прим. 1), на яку вказує `winf`, інформацію про поточні параметри активного текстового вікна та загальні характеристики встановленого відеорежиму.
`gotoxy()`	`void gotoxy (int x, int y);` Переміщує курсор у позицію `x` рядка `y` активного текстового вікна екрана. Використовує віконні координати – (1, 1) відповідає лівому верхньому знакомісцю вікна. Якщо значення координат виходять за межі вікна, то курсор не переміщується.
`highvideo()`	`void highvideo(void);` Встановлює режим підвищеної яскравості для символів, що будуть виводитись на екран наступними операціями консольного виведення (заносить 1 у біт яскравості кольору символа в байті атрибутів).
`insline()`	`void insline(void);` Вставляє порожній рядок, заповнений поточним кольором фону, в активне текстове вікно. Всі наступні рядки вікна зсуваються на один униз. Нижній рядок вікна втрачається. Місце введення порожнього рядка задає поточна позиція курсора, який переноситься на початок вставленого рядка.
`inp()`	`int inp (unsigned port);` Читає один байт з апаратного порту, адресу (номер) якого задає параметр `port`. Повертає зчитаний байт. Реалізовано як макрос.
`inport()`	`unsigned inport (unsigned port);` Читає двобайтове слово з апаратного порту обміну. Молодший байт слова зчитується за адресою `port`, а старший – `port+1`. Повертає зчитане слово. Реалізовано через виклик 16-розрядної команди `in`.
`inportb()`	`unsigned char inportb (unsigned port);` Читає байт з апаратного порту, адресу (номер) якого задає параметр `port`. Повертає значення зчитаного байта.

Функція	Прототип і призначення
`inpw()`	`unsigned inpw (unsigned port);` Читає двобайтове слово з апаратного порту обміну. Молодший байт слова зчитується за адресою `port`, а старший – `port+1`. Повертає зчитане слово. Реалізовано як макрос.
`kbhit()`	`int kbhit (void);` Перевіряє, чи в буфері клавіатури є незчитані символи, тобто чи відбувалось натискання на клавішу. Не зупиняє виконання програми і не зчитує символи з буфера клавіатури. Повертає нуль, якщо буфер клавіатури порожній, та ненульове значення, якщо в буфері є незчитані символи.
`lowvideo()`	`void lowvideo (void);` Встановлює режим зниженої яскравості для символів, що будуть виводитись на екран наступними операціями консольного виведення (скидає в 0 біт яскравості кольору в байті атрибутів).
`movetext()`	`int movetext (int left, int top, int right, int bot, int newleft, int newtop);` Копіює вікно, задане координатами лівого верхнього (`left, top`) та правого нижнього (`right, bot`) кутів, в інше місце екрана, координати лівого верхнього кута якого задають два останні параметри (`newleft, newtop`). Базове вікно зберігається на екрані й залишається активним, поточна позиція текстового курсора не змінюється. Всі координати повноекранні. Повертає ненульове значення, якщо копіювання вікна виконано успішно, інакше (наприклад, якщо нове зображення виходить за межі екрана) вікно не копіюється, а `movetext()` повертає нуль.
`normvideo()`	`void normvideo (void);` Відновлює значення кольорів символів і фону, які були встановлені на момент запуску програми. Впливає на результат наступних операцій консольного виведення.
`outp()`	`int outp (unsigned port, int val);` Передає молодший байт значення, заданого параметром `val`, в апаратний порт обміну `port`. Повертає записане значення. Реалізовано як макрос.
`outport()`	`void outport (unsigned port, unsigned val);` Передає слово, задане параметром `val`, в апаратний порт обміну. Молодший байт слова записується за адресою `port`, а старший – `port+1`. Реалізовано через виклик 16-розрядної команди out.
`outportb()`	`void outportb (unsigned port, unsigned char val);` Передає однобайтове значення параметра `val` в апаратний порт виведення, заданий параметром `port`.
`outpw()`	`unsigned outpw (unsigned port, unsigned val);` Передає слово, задане параметром `val`, в апаратний порт обміну. Молодший байт слова записується за адресою `port`, а старший – `port+1`. Повертає записане слово. Реалізовано як макрос, що викликає 16-розрядну команду out.
`putch()`	`int putch (int sym);` Виводить символ, код якого задає молодший байт параметра `sym`, у поточну позицію активного текстового вікна. Виведення символа `'\n'` тільки переводить курсор на наступний рядок, але не встановлює його на початок цього рядка (див. §16.1.3). Повертає код виведеного символа за умови успішного завершення та макроконстанту `EOF` у разі виникнення помилки.

Функція	Прототип і призначення
`puttext()`	`int puttext (int left, int top, int right, int bot, void* buf);` Відтворює на екрані текстову інформацію з буфера, адресу початку якого задає вказівник `buf`. Вміст буфера виводиться у прямокутну область, задану координатами лівого верхнього (`left, top`) та правого нижнього (`right, bot`) кутів. Нумерація координат повноекранна. Для відображення символів використовуються їх збережені атрибути. Дає змогу відновлювати текстові вікна, попередньо записані функцією `gettext()`. Повертає ненульове значення, якщо координати нового вікна задано коректно й успішно відбулось виведення тексту (див. §16.1.2), інакше вікно не виводиться, а `puttext()` повертає нуль.
`_setcursor-type()`	`void _setcursortype (int cur_t);` Встановлює форму відображення та видимість текстового курсора. Параметр `curs` може приймати одне з трьох значень: `_NOCURSOR` – невидимий курсор, `_SOLIDCURSOR` – високий курсор (повне знакомісце), `_NORMALCURSOR` – звичайний курсор (див. прим. 2).
`textattr()`	`void textattr (int attr);` Встановлює значення атрибутів символа (колір символів, колір фону та режим блимання), задані параметром `attr`. Атрибути можна формувати з числових значень або з іменованих констант переліку `COLORS` (див. прим. 3). Блимання задається додаванням до значення `attr` макроконстанти `BLINK` (див. прим. 4). Встановлене значення атрибутів символа буде використовуватись у всіх наступних функціях консольного виведення.
`textback-ground()`	`void textbackground (int bkcolor);` Встановлює колір фону символів, заданий параметром `bkcolor`. Стандартно для кольору фону в байті атрибутів виділено три біти, тому палітра кольорів фону обмежена значеннями від `0` до `7` (див. §16.1.1). Встановлене значення кольору фону буде використовуватись у всіх наступних функціях консольного виведення.
`textcolor()`	`void textcolor (int symcolor);` Встановлює колір зображення символів, заданий параметром `symcolor`, який може бути іменованою константою з переліку `COLORS` (див. прим. 3) або відповідним числовим значенням. Всі наступні операції консольного виведення даних будуть здійснюватися цим кольором. Якщо до значення кольору додати макроконстанту `BLINK` (див. прим. 4), то виведені на екран символи будуть блимати за умови, що у відеосистемі включено режим блимання, інакше символи будуть відтворюватися з підвищеною яскравістю фону.
`textmode()`	`void textmode (int newmode);` Встановлює текстовий режим, номер якого задає параметр `newmode` (див. §16.1). Режим можна задавати числовим значенням або відповідною іменованою константою з переліку `text_modes` (див. прим. 5).
`ungetch()`	`int ungetch (int ch);` Заносить (повертає) заданий символ `ch` безпосередньо у буфер клавіатури. Наступна операція зчитування з клавіатури введе цей символ першим. Повертати можна тільки один символ, а повторно звертатись до `ungetch()` тільки тоді, коли попередній повернутий символ зчитано. У разі успішного виконання значення функції дорівнює коду записаного символа, а в разі виявлення помилки – `EOF`.

Функція	Прототип і призначення
`wherex()`	`int wherex(void);` Повертає горизонтальну координату текстового курсора (позицію курсора в рядку активного вікна).
`wherey()`	`int wherey(void);` Повертає вертикальну координату текстового курсора (номер рядка активного вікна, на якому розміщений курсор).
`window()`	`void window (int left, int top, int right, int bot);` Встановлює область активного текстового вікна. Координати вікна задаються параметрами: (`left`, `top`) – лівий верхній кут вікна (`left` – горизонтальна, `top` – вертикальна координата); (`right`, `bot`) – правий нижній кут вікна (`right` – горизонтальна, `bot` – вертикальна координата). Координати повноекранні, рядки та стовпчики нумеруються, починаючи з 1 (див. §16.1.2). У разі успішного виконання функції курсор встановлюється у лівий верхній кут активного вікна. Наступні операції консольного виведення будуть відтворювати інформацію в межах встановленого вікна.

Примітки:

1. `struct text_info` – структура, призначена для запису параметрів активного текстового вікна та загальних характеристик встановленого відеорежиму. Склад її полів такий:

```
struct text_info {
   unsigned char winleft;            /* координата лівої межі вікна */
   unsigned char wintop;           /* координата верхньої межі вікна */
   unsigned char winright;           /* координата правої межі вікна */
   unsigned char winbottom;        /* координата нижньої межі вікна */
   unsigned char attribute;       /* атрибут тексту активного вікна */
   unsigned char normattr;             /* початковий атрибут тексту */
   unsigned char currmode;         /* встановлений текстовий режим */
   unsigned char screenheight;       /* вертикальний розмір екрана */
   unsigned char screenwidth;      /* горизонтальний розмір екрана */
   unsigned char curx;   /* поточна горизонтальна координата курсора */
   unsigned char cury;     /* поточна вертикальна координата курсора */
};
```

2. `_NOCURSOR`, `_SOLIDCURSOR`, `_NORMALCURSOR` – макроконстанти, що задають форму відображення та видимість текстового курсора. Їх значення дорівнюють `0`, `1` та `2` відповідно.

3. `enum COLORS` – перелік, іменовані константи якого задають 16-елементну палітру кольорів:

```
enum COLORS {
    BLACK,                  /* темні кольори */
    BLUE,
    GREEN,
    CYAN,
    RED,
    MAGENTA,
    BROWN,
    LIGHTGRAY,
```

```
    DARKGRAY,                   /* світлі кольори */
    LIGHTBLUE,
    LIGHTGREEN,
    LIGHTCYAN,
    LIGHTRED,
    LIGHTMAGENTA,
    YELLOW,
    WHITE
};
```

4. `BLINK` – макроконстанта, що задає блимання символа, її значення дорівнює `128`.
5. `enum text_modes` – перелік, константи якого іменують стандартні текстові режими. Його оголошено так:

```
enum text_modes {
    LASTMODE=-1,
    BW40=0,
    C40,
    BW80,
    C80,
    MONO=7,
    C4350=64
};
```

Додаток 3

НОВЕ В СТАНДАРТІ С-99

Новий стандарт мови С, затверджений у 1999 р., зберіг усі базові концепції та конструктивні принципи, якими вирізняється мова С. Вона й надалі залишатиметься "компактною, прозорою та ефективною" [17]. Доповнення та незначні зміни, введені цим стандартом, відобразилися в трьох основних результатах:

- усунення явних недоліків;
- розширення сфери застосування мови, зокрема в наукових та інженерних розрахунках;
- інтернаціоналізація програм, яка забезпечується використанням універсальних назв символів і підтримкою операцій з широкоформатними символами.

У цьому Додатку стисло викладено всі основні зміни, розширення й доповнення, внесені в мову С стандартом С-99. Їх об'єднано в декілька груп, кожна з яких пов'язана з певними конструктивними компонентами мови. Значна частина нововведень уже згадувалась у відповідних розділах посібника.

Д3.1. Нові ключові слова

Стандарт С-99 ввів у мову п'ять нових службових (ключових) слів:

```
_Bool   _Complex   _Imaginary   inline   restrict
```

Призначення цих слів розглянемо в наступних параграфах.

Д3.2. Розширення щодо коментарів, місця оголошення змінних і граничних рівнів

Однорядкові коментарі. Стандарт С-99 узаконив використання у С-програмах однорядкових коментарів. Цей вид коментарів введено мовою С++, вони дуже зручні для швидкого запису програм. Однорядкові коментарі записують так:

```
. . .          // увесь текст до кінця рядка - коментар
```

Застосування цих коментарів обговорювалось у параграфі 2.3.

Оголошення змінних. Новий стандарт скасував обов'язковість оголошення змінних перед першим оператором у блоці чи в функції. Тепер оголошення змінної можна виконувати в довільному місці програми, але воно повинно передувати першому звертанню до цієї змінної.

```
/* Приклад оголошення змінних у програмному блоці */
{
      double x, y;                  // на початку блоку
   scanf("%lf %lf", &x, &y );
      int q;                        // після оператора
   q = (int)(x/y+0.5);
   . . .
}
```

Також дозволено оголошувати змінні в заголовку оператора циклу `for`. Такі змінні локалізуються у межах блоку, що пов'язаний з цим оператором (відповідний приклад наведено в параграфі 6.4.1).

Збільшення граничних значень. Граничним називають найменше значення відповідного параметра, яке обов'язково повинен підтримувати кожен компілятор. Новий стандарт істотно збільшив низку граничних значень, зокрема:

Кількість	C-89	C-99
значущих символів у внутрішньому ідентифікаторі	31	63
значущих символів у зовнішньому ідентифікаторі	6	31
елементів у структурі або об'єднанні	127	1023
рівнів вкладення блоків	15	127
фактичних параметрів у звертанні до функції	31	127

Д3.3. Нові типи даних і кваліфікатор `restrict`

Стандарт C-99 доповнив мову C декількома новими вбудованими типами та кваліфікатором типу `restrict`. Зупинимось на цих нововведеннях.

Дуже довгі цілі. До складу вбудованих цілих типів мовиС введено два нових: `long long int` та `unsigned long long int`. Це 64-розрядні типи, які забезпечують широкий діапазон значень цілочислових даних (див. табл. 3.1). У записах числових констант, що повинні зберігатись у форматі `long long`, необхідно вказувати суфікс `ll` або `LL`. Наприклад: `42370011`, `037773LL`, `72ull`, `0X1DA72ULL`.

Тип `_Bool`. Цей тип введено для зручності роботи зі змінними, що можуть набувати тільки значення `1` (істина – TRUE) або `0` (хибність – FALSE). У разі присвоєння змінній, що має тип `_Bool`, довільного ненульового значення вона набуде значення `1`, яким позначається істина.

Щоб забезпечити сумісність з типами мови C++, у бібліотеку C-99 включено заголовний файл `<stdbool.h>`, в якому для типу `_Bool` задекларовано найменування `bool` і визначено дві макроконстанти: `true` та `false`.

Комплексні типи. C-99 широко підтримує роботу з комплексними числами. Тому до складу вбудованих типів мови введено ще шість нових:

```
float _Complex      double _Complex      long double _Complex
float _Imaginary    double _Imaginary    long double _Imaginary
```

Перша трійка типів призначена для роботи з комплексними даними. Такі дані зберігаються у формі двох послідовно записаних дійсних чисел відповідного типу, що задають дійсну та уявну частину комплексного числа. Друга трійка типів використовується для збереження лише уявної частини комплексного числа.

У заголовному файлі `<complex.h>` додатково задекларовано найменування типів `complex` та `imaginary` (ці найменування можна застосовувати замість службових слів `_Complex` та `_Imaginary`), а також визначено макроконстанту `I`, значення якої дорівнює $\sqrt{-1}$. Використовуючи ці макровизначення, можна, наприклад, оголосити комплексні змінні наступним чином:

```
double imaginary imgn = 4.85*I;
double complex cmxn = 5.4-imgn;
```

У `<complex.h>` оголошено також набір функцій, які реалізують математичні операції над комплексними числами.

Кваліфікатор `restrict`. Одним із найбільш популярних нововведень С-99 вважається кваліфікатор `restrict` (*restrict* – обмежувати). Його застосовують тільки до вказівників. Вказівник, оголошений з кваліфікатором `restrict`, отримує виключне право звертання до ділянки пам'яті, адресу якої він зберігає. Ніякий інший вказівник чи змінна не можуть звертатись до об'єкта, який адресується `restrict`-вказівником (виняток становлять тільки ті вказівники, що походять від цього `restrict`-вказівника).

Основною сферою застосування кваліфікатора `restrict` є параметри функцій. Розглянемо його роль на прикладі стандартної функції `strcpy()`. У бібліотеках С-89 цю функцію оголошено так:

```
char *  strcpy (char * s1, const char * s2);
```

Додатково словесно зазначають, що ділянки, адреси яких задають вказівники `s1` та `s2`, не повинні перекриватись.

В оголошенні функції `strcpy()` за стандартом С-99 типи параметрів-вказівників доповнено кваліфікатором `restrict`:

```
char *  strcpy (char * restrict s1, const char * restrict s2);
```

Слово `restrict` без додаткових роз'яснень вказує, що кожен із вказівників `s1` та `s2` повинен адресувати окрему ділянку пам'яті.

Компілятор не завжди може повністю проконтролювати дотримання умов, які накладаються кваліфікатором `restrict`. Зокрема, основним призначенням `restrict` у функції `strcpy()` є інформування користувача про обмеження щодо параметрів.

Другим застосуванням кваліфікатора `restrict` є вказівники на об'єкти в динамічній пам'яті. Якщо адресу ділянки динамічної пам'яті занести в `restrict`-вказівник:

```
    double * restrict par;
par = (double *)malloc(sizeof(double) * KEL);
```

то компілятор "знатиме", що до цієї ділянки можна звертатись тільки через `par`. Тому він зможе оптимізувати деякі звертання до елементів масиву в динамічній пам'яті.

Д3.4. Нововведення, пов'язані з масивами та структурами

Стандарт С-99 надав нові можливості щодо оголошення та застосування масивів і структур. Найважливіші з них: створення масивів змінної довжини і гнучких масивів, вибіркова ініціалізація масивів і структур, застосування специфікаторів в оголошеннях параметрів-масивів, формування складених літералів.

Масиви змінної довжини. Такі масиви називають також VLA-масивами (VLA – Variable Length Array – масив змінної довжини). За стандартом С-89 значення розмірності в оголошенні масиву можна задавати тільки константою або константним виразом. Це ускладнює процес програмування, оскільки в багатьох задачах реальна розмірність масиву стає відомою тільки під час виконання програми. Тому стандарт С-99 дозволив задавати розмірність масиву довільним цілочисловим виразом, до складу якого можуть входити змінні програми.

```
/* Приклад оголошення масиву змінної довжини */
int  ConvertElems (int el_numb)
{
      unsigned temp[el_numb];          // VLA-масив
    . . .                              // робота з елементами temp
}
```

Застосування масивів змінної довжини пов'язане з такими обмеженнями та умовами:

- VLA-масиви можуть бути тільки локальними, тобто їх можна оголошувати лише всередині функцій, програмних блоків або в списку формальних параметрів функції;
- змінним, що використовуються у виразах розмірності VLA-масиву, на момент оголошення цього масиву мають бути присвоєні конкретні значення. Наступний приклад ілюструє правильний і хибний порядок запису параметрів у прототипі функції, яка опрацьовує заданий масив змінної довжини.

```
void  InitMtr1 (int rws, int cls, long m[][cls]);     /* правильний
                                                  порядок параметрів функції */
void  InitMtr2 (long m[][cls], int rws, int cls); /* хибний порядок */
```

- після оголошення розмірність масиву змінювати не можна, тобто VLA-масиви не є динамічними;
- у прототипах функцій розмірність VLA-масиву можна позначати символом *:

```
void  InitMtr3 (int, int, long m[][*]);  /* теж правильний прототип */
```

Вибіркова ініціалізація. Часто виникає потреба ініціалізувати окремі елементи масиву або структури. С-99 реалізував таку можливість через конструкції, які називаються *призначеними ініціалізаторами*. Їх синтаксис і дію продемонструємо трьома прикладами:

```
int  vc1[MAX] = {[9]=20, [2]=6};      /* ініціалізація vc1[2] і vc1[9] */
int  vc2[MAX] = {7, 9, 14, 2, [8]=5};       /* ініціалізація елементів
                                              vc2[0]..vc2[3] та vc2[8] */
```

```
struct RGBcolor {
   int red, green, blue;
} colr = {.green=45, .red=28};    // ініціалізація полів green і red
```

Зручною властивістю вибіркової ініціалізації масивів і структур є довільний порядок запису призначених ініціалізаторів.

Гнучкі масиви як елементи структур. Стандарт С-99 дозволив оголошувати останнє поле структури (за умови, що ця структура має інші поля) як масив, розмірність якого не вказується.

```
/* Приклад структури з гнучким масивом */
struct flexible {
   int first;              // визначені елементи структури
   double second;
   char flexstr[];         // гнучкий масив
};
```

Компілятор не виділяє пам'ять для гнучкого масиву, тому значенням операції `sizeof(struct flexible)` буде розмір структури `struct flexible` без останнього поля. Здебільшого такі структури розміщують у динамічній пам'яті, виділяючи додаткову ділянку для елементів гнучкого масиву:

```
    struct flexible *pfs;
pfs = (struct flexible *)malloc(sizeof(*pfs)+flxlen);
```

Після виклику `malloc()` у динамічній пам'яті буде виділено місце для всіх полів структури та ще `flxlen` байт для елементів гнучкого масиву.

Застосування специфікаторів у масивах-параметрах. Оголошуючи масиви в списку параметрів функцій, можна вказувати у квадратних дужках специфікатор `static` або кваліфікатори `const`, `volatile` та `restrict`. Ось приклад:

```
/* Застосування специфікатора static в оголошенні масиву */
void IncrArr (unsigned arr[static 100])
{
    . . .                  // опрацювання елементів масиву arr
}
```

Специфікатор `static` у наведеному прикладі вказує, що масив, адреса якого задається параметром `arr`, буде містити не менше, ніж 100 беззнакових елементів.

Кваліфікатор `const`, записаний в оголошенні масиву, зазначає, що у викликах функції це буде один і той самий масив. Кваліфікатор `volatile` вказує, що це масив, який може асинхронно змінюватись зовнішнім середовищем, а кваліфікатор `restrict` встановлює, що тільки цей вказівник (ім'я масиву) можна застосовувати для звертання до елементів масиву.

Складені літерали. *Складені літерали* – це набори констант, сформовані як масиви, структури або об'єднання. Їх введено в С насамперед, щоб забезпечити можливість передавати в функцію для опрацювання відповідні складені константи.

Синтаксис складених літералів такий:

(найменування_типу) { список значень_елементів }

Наведемо приклад створення складеного літерала-масиву, адресу початку якого зберігатиме вказівник `pcar`:

```
int *pcar = (int[]){11,18,21};
```

Фактично `pcar` буде вказувати на масив із трьох цілих чисел, які мають задані значення.

Важливо, що складені літерали можна використовувати як фактичні параметри в звертаннях до функцій. Якщо в програмі оголошено функцію

```
void  PrintArray (unsigned arr[], int n);
```

то звертання до неї може бути таким:

```
PrintArray ((unsigned[]){1,2,3,5,7,11,13}, 7);
```

Складені літерали можуть бути також структурами або об'єднаннями. Проілюструємо це наступним прикладом:

```
/* Застосування складеного літерала-структури */
typedef struct box {           // шаблон структури
          int x1, y1, x2, y2;
        } BOX;
long  Square (BOX rect)        // функція обчислення площі
{
   return (long)(rect.x2-rect.x1+1)*(rect.y2-rect.y1+1);
}
   long newsq;
newsq = Square((BOX){20,1, 65,14});
```

Д3.5. Зміни й доповнення в програмуванні функцій

Стандарт С-99 скасував усі неоднозначності в оголошеннях функцій та їх прототипів. Затверджено три вимоги, подані нижче.

Скасування "неявного `int`". По-перше, в оголошенні функції тип її значення та типи всіх параметрів мають бути вказані явно. Нагадаємо, що згідно з попередніми стандартами, якщо тип значення, яке повертала функція, або тип якогось із параметрів був цілочисловим, то слово `int` в оголошенні такої функції можна було опускати (ці питання розглядались детальніше в параграфі 2.1).

Вирази в операторі `return`. Друга вимога С-99 стосується оператора повернення з функції `return`. Якщо тип значення функції відмінний від `void`, то в операторі `return` обов'язково має бути записаний вираз, значення якого повертає ця функція (раніше такий вираз дозволялось опускати).

Обов'язковість прототипу. Третя вимога: кожна функція повинна бути явно оголошена або описана перед першим звертанням до неї. Неявні оголошення функцій скасовано.

Службова змінна `__func__`. Цю спеціальну змінну можна застосовувати всередині функції, щоб отримати ім'я цієї функції у формі внутрішнього символьного рядка. Розглянемо приклад:

```
/* Застосування __func__ */
void  Polygon (int nv)
{
     int ctr;
  . . .                              // початкова частина функції
  if (ctr<0)
    printf ("Функція %s() => помилка в даних!", __func__);
  . . .                              // продовження функції
}
```

У функції `Polygon()` використано службову змінну `__func__`, щоб включити в повідомлення про помилку найменування поточної функції. Якщо значення локальної змінної `ctr` від'ємне, то на екран виводиться текст:

```
Функція Polygon() => помилка в даних!
```

Кожна зміна імені функції автоматично відобразиться в тексті повідомлення про помилку.

Функції з кваліфікатором `inline`. Ключове слово `inline`, записане на початку заголовка функції, вказує компілятору про бажання користувача оптимізувати код цієї функції за швидкодією. Детальніше `inline`-функції розглянуті в параграфі 11.4.

Д3.6. Розширення бібліотечних функцій і бібліотек

Стандарт С-99 приділив значну увагу бібліотекам мови С. По-перше, доповнено деякі з наборів функцій, що підтримувались стандартом С-89. Так, істотно збільшилась бібліотека `<math.h>`, в яку введено ряд нових функцій, а також розширено наявні функції для операцій з даними, що мають тип `float` і `long double`. По-друге, у бібліотечні функції внесено зміни згідно з нововведеннями, затвердженими у С-99 (наприклад, доповнено кваліфікатором `restrict` відповідні параметри-вказівники). По-третє, створено бібліотеки для нових категорій даних, макросів і функцій. Охарактеризуємо коротко запроваджені розширення бібліотек.

Нові специфікатори форматів для функцій `printf()` і `scanf()`. Для форматного введення/виведення числових даних, що мають типи `long long int` та `unsigned long long int`, встановлено модифікатор `ll`. Цей модифікатор можна застосовувати з усіма специфікаціями цілочислових форматів: `d`, `i`, `u`, `o`, `x`/`X`. Наведемо приклад введення значення дуже довгого цілого:

```
    long long int  dist;
scanf("%lld", &dist);
```

Ще один новий модифікатор – `hh` призначений для операцій числового введення/виведення даних, що мають тип `char`.

Нові специфікації формату `a` та `A` дають змогу виводити (або вводити) шістнадцяткові дані у форматі чисел з плаваючою крапкою:

`[-]0x`*h.hhhh*`p+`*d* – у разі специфікації `a`;

`[-]0X`*h.hhhh*`P+`*d* – у разі специфікації `A`.

Нові бібліотеки функцій та заголовні файли. Стандартом С-99 затверджено декілька нових бібліотек і відповідних заголовних файлів. Назвемо їх:

Заголовний файл		Призначення бібліотеки
`<complex.h>`	–	реалізація арифметичних операцій та математичних функцій над комплексними числами;
`<fenv.h>`	–	керування операціями з плаваючою крапкою через набір прапорців і режимів роботи обчислювального середовища;
`<inttypes.h>`	–	перетворення форматів цілочислових даних у значення заданої розрядності;
`<iso646.h>`	–	встановлення набору макроімен для порозрядних і логічних операцій (наприклад: `&` – `bitand`, `&&` – `and`, `^` – `xor` тощо);
`<stdbool.h>`	–	декларування типу `bool` і визначення макроконстант `true` та `false`, чим забезпечується сумісність з С++;
`<stdint.h>`	–	декларування найменувань т. зв. розширених типів;
`<tgmath.h>`	–	встановлення макросів т. зв. абстрактних дійсних чисел для реалізації певних математичних функцій;
`<wchar.h>`	–	введення/виведення та опрацювання дво- та багатобайтових символів і символьних рядків;
`<wctype.h>`	–	класифікація та перетворення т. зв. широких символів.

Розширені цілочислові типи. У заголовному файлі `<stdint.h>` через `typedef` задекларовано низку найменувань спеціальних цілих типів, використання яких сприяє високій мобільності С-програм. Ці типи називають *розширеними* цілими типами. Всі найменування розширених типів можна поділити на декілька груп:

`int`*B*`_t`	–	цілий тип, розрядність якого дорівнює *B*;
`uint`*B*`_t`	–	тип цілого без знака, що має розрядність *B*;
`int_least`*B*`_t`	–	цілий тип, розрядність якого не менша за *B*;
`int_fast`*B*`_t`	–	цілий тип, що забезпечує найвищу швидкодію і має розрядність не меншу за *B*;
`int_max_t`	–	цілий тип, що має найбільшу допустиму для даного середовища розрядність;
`uint_max_t`	–	цілий беззнаковий тип, що має найбільшу допустиму для даного середовища розрядність.

Літерою *B* в узагальнених найменуваннях розширених типів позначено розрядність відповідного типу, вона може становити 8, 16, 32 або 64 біти. Наприклад, `int_fast32_t` – це найбільш швидкий із цілих типів, розмір яких не менший за чотири байти.

У `<stdint.h>` визначено також два макроси з параметрами:

`int`*B*`_C (`*вираз-параметр*`)`

`uint`*B*`_C (`*вираз-параметр*`)`

Результатом застосування цих макросів є число, яким можна записати значення виразу-параметра в формі *B*-розрядного цілого зі знаком чи без знака відповідно.

Операції з широкими символами. До складу бібліотек С-99 включено нові заголовні файли `<wchar.h>` і `<wctype.h>`, макроси та функції яких дають змогу використовувати в С-програмах т. зв. *широкі символи* (*wide-characters*). Ці символи введено в мову С у 1995 р. (Поправка 1) для підтримки багатолітерних національних мов, насамперед азійських.

Функції, що опрацьовують широкі символи, оперують з типом `wchar_t`, задекларованим через `typedef` (найчастіше він встановлюється рівнозначним двобайтовому цілому типу `unsigned short int`). Крім `wchar_t` задекларовано допоміжні типи `wint_t`, `wctrans_t` та `wctype_t`.

Бібліотека, пов'язана зі заголовним файлом `<wchar.h>`, зберігає великий набір функцій. Вони забезпечують введення та виведення широких символів, реалізують операції над рядками та масивами широких символів (подібні до операцій, що оголошені в `<string.h>` і `<mem.h>`), виконують перетворення двобайтових символів і рядків широких символів у багатобайтові рядки, а також реалізують ряд інших операцій.

У заголовному файлі `<wctype.h>` записано прототипи функцій, які призначені для класифікації та перетворення широких символів. Ці функції аналогічні до функцій, оголошених у `<ctype.h>`. З переліком функцій опрацювання широких і багатобайтових символів можна ознайомитися в [24].

Список літератури

1. Белецкий Я. Энциклопедия языка Си: Пер. с польск. – М.: Мир, 1992. – 686 с.
2. Березин Б.И., Березин С.Б. Начальный курс С и С++. – М.: Диалог-МИФИ, 1998. – 288 с.
3. Бочков С. О. Язык программирования Си для персонального компьютера. – М.: Диалог, 1990. – 384 с.
4. Вирт Н. Алгоритмы и структуры данных: Пер. с англ. – СПб.: Невский диалект, 2001. – 352 с.
5. Джамса К. 1001 совет по C/C++. Настольная книга программиста: Пер. с англ. – М.: Март-Бином Универсал, 1997. – 784 с.
6. Джонс Б., Эйткен П. Освой самостоятельно С за 21 день. Пер. с англ. – М.: Изд. дом "Вильямс", 2003. – 800 с.
7. Касаткин А.И., Вальвачев А.Н. Профессиональное программирование на языке Си. От Turbo C к Borland C++. Справочное пособие. – Минск: Вышэйшая школа, 1992. – 240 с.
8. Касаткин А.И., Вальвачев А.Н. Профессиональное программирование на языке Си. Управление ресурсами. – Минск: Вышэйшая школа, 1992. – 432 с.
9. Керниган Б., Пайк Р. Практика программирования: Пер. с англ. – СПб.: Невский диалект, 2001. – 381 с.
10. Керниган Б., Ритчи Д. Язык программирования Си: Пер. с англ. – М.: Финансы и статистика, 1992. – 272 с.
11. Керниган Б., Ритчи Д. Язык программирования С: Пер. с англ. – Изд. дом "Вильямс", 2005. – 304 с.
12. Кормен Т. и др. Алгоритмы: построение и анализ: Пер. с англ. – Изд. дом "Вильямс", 2005. – 1296 с.
13. Кнут Д.Е. Искусство программирования. Т.2. Получисленные алгоритмы: Пер. с англ. – М.: Изд. дом "Вильямс", 2004. – 824 с.
14. Кнут Д.Е. Искусство программирования. Т.3. Сортировка и поиск: Пер. с англ. – М.: Изд. дом "Вильямс", 2004. – 780 с.
15. Подбельский В.В., Фомин С.С. Программирование на языке Си. – М.: Финансы и статистика, 1999. – 660 с.
16. Подбельский В.В. Практикум по программированию на языке Си. – М.: Финансы и статистика, 2004. – 576 с.
17. Прата С. Язык программирования С. Лекции и упражнения: Пер. с англ. – СПб.: ООО "ДиаСофт ЮП", 2002. – 896 с.
18. Тондо К., Гимпел С. Язык Си. Книга ответов: Пер. с англ. – М.: Финансы и статистика, 1994. – 157 с.
19. Трой Д. Программирование на языке Си для персонального компьютера IBM PC: Пер. с англ. – М.: Радио и связь, 1991. – 432 с.
20. Троценко В.С., Чаленко П.Й., Ставровський А.Б. Техніка програмування мовою Сі. К.: Либідь, 1993. – 224 с.
21. Уинер Р. Язык Турбо Си: Пер. с англ. – М.: Мир, 1991. – 384 с.
22. Уэйт Р., Прата С., Мартин Д. Язык Си. Руководство для начинающих: Пер. с англ. – М.: Мир, 1988. – 512 с.
23. Хэзфилд Р., Кирби Л. и др. Искусство программирования на Си. Фундаментальные алгоритмы, структуры данных и примеры приложений. Энциклопедия программиста: Пер. с англ. – К.: Изд-во "ДиаСофт", 2001. – 736 с.
24. Шилдт Г. Полный справочник по С: Пер. с англ. – М.: Изд. дом "Вильямс", 2002. – 704 с.
25. Юлин В.А., Булатова И.Р. Приглашение к Си. – Минск: Вышэйшая школа, 1990. – 224 с.

Предметний покажчик

Лексеми

Навчальне видання

Шпак Зореслава Ярославівна

Програмування мовою С

Рекомендовано Міністерством освіти і науки України

Видано за рахунок державних коштів. Продаж заборонено

Редактор *Р.Я. Ступницький*
Коректор *З.Я. Сорока*
Обкладинка *С.І. Іванов*
Комп'ютерне верстання *Н.М. Хомуляк*

Здано на складання 07.06.2006. Підписано до друку 02.08.2006.
Формат 70×100/16. Гарнітура Times New Roman. Папір офс. Друк офс.
Умовн. друк. арк. 35,1. Обл.-вид. арк. 32,00. Тираж 4040.
Свідоцтво держ. реєстру ДК № 63. Вид № 03. Зам. № 228П.

Видавництво "Оріяна-Нова"
79017 м. Львів, вул. Коциловського, 10

Віддруковано з готових діапозитивів
у Відкритому акціонерному товаристві "Патент"
88006 м. Ужгород, вул. Гагаріна, 101
Тел.: (0312) 66-07-03; факс: (0312) 66-02-22
e-mail: patent@uzh.ukrtel.net