Мова програмування C

дуге видання

Браян В. Керніган, Деніс М. Річі

Зміст

Передмова

Комп'ютерний світ пережив революцію з моменту публікації «Мови програмування С» у 1878-у році. Комп'ютери стали набагато більші, а особисті комп'ютери включають можливості, що суперничають з універсальними ЕОМ десятиліття тому. Впродовж цього часу, С також змінилася, навіть якщо й помірно, і поширилась далеко за межі свого походження, як мови операційної системи Unix. Зріст популярності С, зміни в мові впродовж цих років і створення компіляторів групами людей, непричетних до розробки самої мови — все це разом вимагає точнішого і сучаснішого визначення мови, аніж те, яке було надано першою публікацією цієї книжки. У 1983-у році, Американський Національний Інститут Стандартів (ANSI) заснував комітет, чиєю ціллю було «недвозначний, машинонезалежний опис мови С», одночасно зберігаючи її основний дух. Як наслідок, з'явився стандарт ANSI С.

Стандарт формалізує конструкції, на які робився натяк, але не описано у першому виданні, зокрема: присвоєння структур та енумерація. Він надає нову форму оголошенням функцій і дозволяє перехресну перевірку оголошень і використання. В ньому описано стандартну бібліотеку з широким набором функцій для здійснення вводу та виводу, керування пам'яттю, маніпулювання ланцюжками та схожих завдань. Він уточнює поводження властивостей, які не було до кінця пояснено у першому виданні, одночасно ясно заявляючи, які аспекти мови залишаються машинозалежними.

Це друге видання «Мови програмування С» описує мову згідно зі стандартом ANSI. Хоч ви знайдете позначеним ті місця, де мова еволюціонувала, ми вирішили написати все дотримуючись нової форми. У більшості випадків, різниця невелика; найочевидніша зміна, це нова форма оголошення функцій та їхнього означення. Сучасні компілятори вже підтримують більшість нововведень стандарту.

Ми намагалися зберегти стислість першого видання. С не є великою мовою і велика книжка не зробить їй хорошої послуги. Ми поліпшили висвітлення критично-важливих рис, таких як покажчики, наприклад, які є центральними для програмування на С. Ми вдосконалили оригінальні приклади і додали нові в декількох розділах. Приміром, розгляд складних оголошень супроводжується програмами, що перекладають оголошення у звичайні слова, і навпаки. Як і раніше, всі приклади було перевірено прямо з тексту, який зберігається у машинопрочитному вигляді.

Додаток А — довідковий посібник — не є самим стандартом, а радше нашим намаганням стисло передати основні риси стандарту. Розділ задумано для легкого розуміння програмістами, а не як визначення для розробників компіляторів — ця роль належить

самому стандартові. Додаток Б — це підсумок можливостей стандартної бібліотеки. Також малося на увазі легке розуміння програмістами, а не розробниками бібліотек. Додаток В — це короткий підсумок змін від оригінальної версії книжки.

Як ми вже сказали у передмові до першого видання, «С краще пасує, як ваш досвід щодо неї зростає». Маючи десятиліття або більше досвіду, ми й досі так вважаємо. Сподіваємося, що ця книжка допоможе вам навчитися С і використовувати її належним чином.

Ми щиро вдячні друзям, хто допоміг нам створити це друге видання. Джону Бентлі, Дагу Ґвіну, Дагу Мак-Ілрою, Пітеру Нельсону і Робу Пайку, які надали нам важливі коментарі для майже кожної сторінки чорнового рукопису. Ми вдячні за уважне прочитання Елові Ахо, Денісу Аллісону, Джо Кембелу, Ґ.Р. Емліну, Карен Фортган, Аллену Голуб, Ендрю Хьюму, Дейву Крістолу, Джону Ліндерману, Дейву Проссеру, Джину Спаффорду і Крісу ван Віку. Ми також отримали корисні поради від Біла Чесвіка, Марка Кенікгана, Енді Кьонинга, Робина Лейка, Тома Лондона, Джима Рідза, Кловиса Тондо та Пітера Вейнберга. Дейв Проссер відповів на багато детальних запитань щодо стандарту ANSI. Ми скористалися програмою-перекладачем C++ Бжорна Строустрапа для локального тестування наших програм, і Дейв Крістол забезпечив нас компілятором ANSI C для остаточного тестування. Річ Дречслер істотно допоміг з набиранням тексту.

Наша щира подяка всім.

Браян В. Керніган
Деніс М. Річі

Передмова до першого видання

C — це мова програмування загального призначення, що включає економію представлення, сучасне керування потоком і структурою даних і багатий набір операторів. C не є мовою «дуже високого рівня», ні «великою» мовою, і не призначена для певної області застосування. Але відсутність в ній обмежень та її загальність роблять її зручнішою і ефективнішою для багатьох завдань, у порівнянні з мовами, що вважаються потужнішими. Початково, C розроблено та втілено на операційній системі Unix на DEC PDP-11 Деніса Річі. Операційна система, компілятор C і, по суті, всі програми-додатки Unix (включаючи програмне забезпечення, використовуване для приготування цієї книжки) написано на C. Робочі компілятори існують також і для декількох інших машин, включаючи IBM System/370, Honeywell 6000 та Interdata 8/32. Проте, C не прив'язана до певного обладнання чи системи, і на ній легко писати програми, що працюватимуть без змін на будь-якій машині, яка підтримує C.

Ця книжка має на меті допомогти читачеві навчитися програмувати на C. Вона містить вступну частину для ознайомлення нових користувачів, окремі розділи, присвячені основним властивостям мови, та довідник. Процес навчання основується на читанні, складанні програм і перегляді прикладів, замість простого викладу правил. У більшості випадків, приклади — це справжні програми, а не окремі фрагменти коду. Всі приклади перевірено безпосередньо з тексту, який зберігається в машинопрочитному вигляді. Крім демонстрації того, як ефективно користуватися мовою, ми також спробували, де можливо, показати корисні алгоритми, принципи хорошого стилю та правильної розробки.

Ця книжка не є ввідним посібником з програмування, вона передбачає певне знайомство з основними поняттями програмування, такими як змінні, вирази присвоєння, цикли та функції. Не зважаючи на це, програміст-новачок повинен бути спроможним в процесі читання вивчити мову, хоча доступ до досвідченіших колег не завадить.

З нашого досвіду, C зарекомендувала себе як приємна, виразна й різностороння мова для широкого спектру програм. Її легко вивчити і вона краще пасує, як ваш досвід із нею зростає. Ми сподіваємося, що дана книжка допоможе вам використовувати її належним чином.

Продуманий критицизм і поради від чисельних друзів і колег багато додали до цієї книжки і нашого задоволення щодо її написання. Зокрема, Майк Бьянкі, Джим Блю, Стю Фельдман, Даг Мак-Ілрой, Білл Роом, Боб Рисин і Ларрі Рослер, усі вони старанно прочитали значний об'єм рукопису. Ми також завдячуємо Елові Ахо, Сті-

вові Борну, Данові Двораку, Чакові Гарлі, Деббі Гарлі, Меріон Гаррис, Ріков Гольту, Стівові Джонсону, Джонові Меші, Бобові Митзеві, Ральфові Мусі, Пітеру Нельсону, Елліоту Прінсону, Біллу Плагеру, Джеррі Співак, Кену Томсону і Пітеру Вейнбергеру за корисні поради в різноманітних стадіях рукопису, так само як Майку Леску і Джо Оссанні за неоціненну допомогу в набиранні тексту.

Браян В. Керніган
Деніс М. Річі

Вступ

...

Розділ 1

Вступний урок

Почнемо зі швидкого введення в C. Наша мета — показати основні елементи мови в справжніх програмах, уникаючи при цьому надміру деталей, правил та виключень. У цьому розділі ми не збираємось бути вичерпними або точними (за винятком прикладів, які мають бути правильні). Ми хочемо привести вас якнайшвидше до того пункту, де ви зможете самостійно писати корисні програми, і щоб досягти цього, ми повинні зосередитись на базових речах: змінних і константах (сталих), арифметиці, керуванню потоком виконання, функціях і найпростіших операціях вводу та виводу. Ми навмисне залишаємо поза увагою в цьому розділі риси C, важливі для написання більших програм. Це стосується покажчиків, структур, більшості з широкого набору операторів C, декількох виразів керування потоком і, нарешті, стандартної бібліотеки.

Цей підхід має свої недоліки. Найважливіший — це те, що ви тут не знайдете повного опису певної риси мови, і через стислість цей розділ може вводити в оману. Також, з-за того, що в прикладах не застосовується вся потужність мови, вони не настільки стислі та вишукані, якими насправді могли би бути. Ми намагалися звести до мінімуму цей негативний ефект, але все ж попереджаємо вас. Ще одним недоліком є те, що пізніші розділи змушені будуть повторити дещо з пройденого тут. Ми сподіваємося, що повторення скоріше допоможуть вам, аніж дратуватимуть.

В усякому разі, досвідчені програмісти повинні вибрати дещо з цього розділу для власних потреб, тоді як новачкам радимо пройти його повністю, одночасно пишучи подібні власні короткі програми. Обидві групи можуть скористатися з цього розділу як з каркасу, на який додасться детальніший опис пізніше, починаючи з Розділу 2.

1.1 Перші кроки

Єдиний спосіб навчитися нової мови програмування — це власне писати на ній програми. Перша програма для написання — подібна в усіх мовах: вивести на екран

```
hello, world
```

Це є першою, важливою віхою. Щоб її подолати, ви повинні спочатку створити текст програми, компілювати його, завантажити, запустити програму і, нарешті, з'ясувати куди саме надійшов вивід. Після освоєння цих механічних деталей, все інше відносно легко.

У С, програма виводу «hello, world» виглядатиме так:

```
#include <stdio.h>

main()
{
    printf("hello, world\n");
}
```

Як запустити цю програму, певною мірою, залежить від використовуваної вами системи. Так, скажімо, в операційній системі Unix, ви повинні створити файл із закінченням «.c», наприклад `hello.c`, потім компілювати його командою

```
cc hello.c
```

Якщо ви ніде не схибили, як от пропустили якийсь знак або припустилися орфографічної помилки, процес компіляції пройде безшумно і видасть виконуваний файл з назвою `a.out`. Після того, як ви запустите `a.out` командою

```
a.out
```

у вас на екрані з'явиться

```
hello, world
```

На інших системах правила компіляції можуть у чомусь відрізнятися — спитайтеся місцевого фахівця.

Тепер — до пояснення самої програми. Програма мовою С, незалежно від свого розміру, складається з функцій та змінних. Функції містять вирази, що вказують на обчислювальні дії, які матимуть місце, а змінні, в свою чергу, зберігають значення, використовувані під час обчислень. Функції С подібні до підпрограм і функцій Fortran або процедур та функцій Pascal. У нашому прикладі, це функція під назвою `main`. Зазвичай, ви можете надавати своїм функціям довільні імена, але «`main`» — це спеціальна назва. Програма починає своє виконання на початку `main`. Це означає, що кожна програма повинна містити `main` в якомусь місці.

Функція `main` звичайно викликає інші функції для допомоги виконання своєї роботи, деякі написані вами, а деякі з наданих вам бібліотек. Перший рядок програми,

`#include <stdio.h>`	*включає інформацію про стандартну бібліотеку*
`main()`	*означує функцію із назвою* `main`*, яка не отримує жодних значень аргументів*
`{`	*вирази, які належать* `main`*, взято у фігурні дужки*
`printf("hello, world\n");`	*щоб вивести послідовність знаків,* `main` *викликає функцію* `printf` *зі стандартної бібліотеки*
`}`	`\n` *означає знак нового рядка*

Перша С-програма

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

вказує компілятору включити інформацію про стандартну бібліотеку вводу/виводу; цей рядок з'являється на початку багатьох вихідних текстів С. Стандартна бібліотека описана в Розділі 7 і Додатку Б.

Один із способів обміну даними між функціями — це передання викликовою функцією списку значень, які називаються *аргументами*, функції, яку вона викликала. Дужки після назви функції оточують список аргументів. У цьому прикладі `main` вказано як функцію, яка не очікує жодних аргументів, що видно з порожнього списку в дужках `()`.

Твердження, які належать функціям, включаються у фігурні дужки `{ }`. Функція `main` містить тільки одне таке твердження:

```
printf("hello, world\n");
```

Функція викликається вказівкою її назви, за якою слідує список аргументів у дужках, а отже тут викликано функцію `printf` з аргументом «`hello, world\n`». `printf` — функція бібліотеки для друкування виводу, у цьому випадку — ланцюжка знаків між лапками.

Послідовність знаків, включених у подвійні лапки, як от «`hello, world\n`», називається *символьним ланцюжком* або *символьною константою*. Поки-що, єдине місце, де ми будемо вживати символьні ланцюжки, — це в якості аргументу `printf` та інших функцій.

Послідовність знаків `\n` означає в С *символ нового рядка*, який під час виводу переводить ланцюжок тексту на новий рядок. Якщо ви пропустите `\n` (варте того, щоб зробити експеримент), то виявите, що після виводу перенесення рядка не відбувається. Ви маєте скористатися з `\n`, щоб включити знак нового рядка в аргумент `printf`; якщо ж ви спробуєте щось на зразок

```
printf("hello, world
");
```

то компілятор C повідомить про помилку.

`printf` ніколи не додає знака нового рядка автоматично, що дає можливість використання декількох викликів для поетапної побудови виводу. Так, нашу першу програму можна би було з таким самим успіхом написати як

```
#include <stdio.h>

main()
{
    printf("hello, ");
    printf("world");
    printf("\n");
}
```

що призведе до тотожного виводу.

Замітьте, що `\n` насправді означає тільки один знак. *Екрановані послідовності* на кшталт `\n` являють собою загальний механізм позначення «важких для друкування» або невидимих символів. Наряду з іншими, C також передбачає `\t` для табуляції, `\b` для реверсу, `\"` для виводу подвійних лапок і `\\` для самої зворотньої похилої. Повний список можна знайти у Розділі 2.3.

Вправа 1-1. Виконайте програму «`hello world`» на вашій системі. Поекспериментуйте з видаленням частин програми, щоб побачити, які помилки при цьому виникнуть.

Вправа 1-2. Дізнайтесь, що станеться, якщо вказати як аргумент `printf` послідовність `\c`, де `c` є одним із знаків, не згаданих вище.

1.2 Змінні й арифметичні вирази

Програма далі використовує формулу °C=(5/9)(°F-32) для виводу наступної таблиці температур по Фаренгейту та відповідних значень за Цельсієм:

```
1       -17
20      -6
40      4
60      15
80      26
100     37
120     48
140     60
```

```
160       71
180       82
200       93
220       104
240       115
260       126
280       137
300       148
```

Сама програма все ще складається з однієї тільки функції `main`. Вона довша, ніж та, яка виводила «`hello world`», але не є складною. Ця програма привносить декілька нових понять, таких як коментарі, оголошення, змінні, арифметичні вирази, цикли та форматований вивід.

```
#include <stdio.h>

/* вивести таблицю Фаренгейт-Цельсій із fahr = 0, 20, ..., 300 */
main()
{
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;     /* нижня межа температурної шкали */
    upper = 300;   /* верхня межа */
    step = 20;     /* розмір поступу */

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf("%d\t%d\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

Рядок

```
/* вивести таблицю Фаренгейт-Цельсій із fahr = 0, 20, ..., 300 */
```

є *коментарем*, який у цьому випадку коротко пояснює, що саме програма здійснює. Будь-які знаки між `/*` та `*/` ігноруються компілятором. Коментарі можуть вільно використовуватись, щоб зробити програму зрозумілішою. Коментарі можна помістити в будь-якому місці, де можуть знаходитись пробіли, табуляція або знаки нового рядка.

У C всі змінні слід оголосити до того, як користуватися ними — як правило, на початку функції, перед виконанням інших тверджень. В оголошеннях змінних вказуються їхні властивості; кожне оголошення складається з назви типу та списку змінних, на зразок

```
int fahr, celsius;
int lower, upper, step;
```

Тип `int` означає, що ці змінні — цілі числа, на відміну від `float`, що позначає числа з рухомою точкою, тобто числа, які можуть мати дробову частину. Обсяг обох, `int` і `float`, залежить від машини; поширеним типом є 16-бітні значення `int`, що знаходяться у межах від -32768 до +32768, так само як і 32-бітні `int`. Значення `float`, як правило, мають 32-бітну довжину, здатну утримувати шестизначні числа в діапазоні між 10^{-38} і 10^{38}. C передбачає декілька додаткових типів даних, окрім `int` та `float`, включаючи:

`char`	символ — один байт
`short`	коротке ціле число
`long`	довге ціле число
`double`	число з рухомою точкою подвійної точності

Розмір цих даних також є машинозалежним. Окрім цього, існують ще масиви, структури та сполуки з цих основних типів, покажчики на них, а також функції, які повертають їх. Усе це ми пройдемо в слушну мить.

Обчислення в програмі перетворення температур починається з виразів присвоєння

```
lower = 0;
upper = 300;
step = 20;
```

які встановлюють змінні в їхні початкові значення. Кожний окремий вираз повинен закінчуватися крапкою з комою.

Кожний рядок таблиці перетворень обчислюється у той самий спосіб, тож ми використали цикл, що повторюється по одному разу для кожного рядка виводу; саме у цьому полягає зміст циклу `while`

```
while (fahr <= upper) {
    ...
}
```

Цикл `while` діє наступним чином: умова в дужках перевіряється, якщо вона є істинною (значення `fahr` дійсно менше або рівне `upper`), виконується тіло циклу (три

вирази, включені у фігурні дужки). Після цього умова перевіряється знову, якщо істина — тіло буде виконано знову. Коли умова виявиться хибною (`fahr` стане більше за `upper`), цикл завершиться, і виконання програми продовжиться з виразу, що слідує одразу за циклом. Якщо додаткових виразів немає, програма завершиться.

Тіло циклу `while` може складатися з одного або більше тверджень, включених у фігурні дужки, як у програмі перетворення температур, або тільки одного твердження без фігурних дужок, як наприклад

```
while (i < j)
    i = 2 * i;
```

В обох випадках, ми завжди зміщуватимемо праворуч на один крок табуляції вираз, контрольований `while`, щоб було зрозуміло з першого погляду, які вирази знаходяться всередині циклу. Відступи підкреслюють логічну структуру програм. Хоч компілятори C і не зважають на те, як програма виглядає, належні відступи праворуч і пробіли важливі для для прочитності програм. Ми радимо використовувати лише по одному твердженню на рядок і пробіли навколо операторів для ясності стосовно групування. Місцезнаходження фігурних дужок — не настільки важливе, незважаючи на те, що дехто притримується палких переконань з цього приводу. Ми вибрали один з декількох популярних стилів. Зупиніться на тому стилі, який вам найбільше до вподоби, і дотримуйтеся його.

Найбільше роботи здійснюється в тілі циклу. Температура за Цельсієм обчислюється та зберігається у змінній `celsius` виразом

```
celsius = 5 * (fahr-32) / 9;
```

Причиною множення на п'ять а потім ділення на 9 замість просто множення на $5/9$ є те, що C, як і багато інших мов, стинає результат поділу цілих чисел — дробова частина відкидається. Оскільки 5 і 9 — обидва цілі числа, поділ $5/9$ округлиться до нуля, тож всі температури за Цельсієм звітуватимуться як нульові.

Цей приклад програми також розкриває нам трохи більше стосовно роботи `printf`. `printf` — це функція загального призначення для форматованого виводу (ми її опишемо докладніше в Розділі 7). Її першим аргументом є ланцюжок знаків, які буде виведено, де `%` вказує ті частини, які буде замінено, і в якій формі відбудеться вивід. Так наприклад, `%d` вказує на десятковий аргумент, тож вираз

```
printf("%d\t%d\n", fahr, celsius);
```

виведе значення двох цілих `fahr` і `celsius`, розділені табуляцією (`\t`).

Кожна конструкція з `%` першого (включеного у лапки) аргументу `printf`, знаходить собі пару у другому, третьому, і так далі, аргументові `printf`; вони мають збігтися кількісно і за типом, інакше ви отримаєте помилкові відповіді.

Між іншим, `printf` не є частиною мови C, ввід і вивід не визначено у самій мові. `printf` — це просто функція серед інших функцій стандартної бібліотеки мови. Однак, поводження printf описане стандартом ANSI, тож воно має бути однаковим, незалежно від компілятора та платформи, які використовуються.

Для того, щоб зосередитись на самій C, ми не обговорюватимемо ввід і вивід значною мірою аж до Розділу 7. Зокрема, ми втримаємося до того часу від форматованого вводу. Якщо вам треба написати щось із вводом чисел, зазирніть до обговорення функції scanf у Розділі 7.4. Остання схожа на `printf`, за виключенням того, що `scanf` читає ввід, замість здійснювати вивід.

Наша програма перетворення температури має проте декілька вад. Найпростіша з них це те, що вивід — не досить привабливий, оскільки числа не вирівняно з правого боку. Це легко виправити, якщо додати до кожного твердження `%d` функції `printf` аргумент ширини. У цьому випадку, числа буде вирівняно праворуч. Так, наприклад, ми можемо сказати

```
printf("%3d %6d\n", fahr, celsius);
```

щоб вивести перше число кожного рядка шириною в три знаки, а друге число — шириною в шість:

```
  0     -17
 20      -6
 40       4
 60      15
 80      26
100      37
...
```

Серйозніша проблема полягає у тому, що ми використали арифметику десяткових чисел, тож отримана температура за Цельсієм — не досить точна. Наприклад, 0F, насправді дорівнює -17.8C, а не -17. Щоб отримати точніші відповіді, ми повинні звернутися до арифметики чисел з рухомою точкою замість цілих. Це вимагає деяких змін у програмі. Ось друга версія:

```
#include <stdio.h>

/* виводить таблицю Фаренгейт-Цельсій із fahr = 0, 20, ..., 300;
   версія з числами з рухомою точкою */
main()
{
    float fahr, celsius;
    float lower, upper, step;
```

```
    lower = 0;     /* нижня межа температурної шкали */
    upper = 300;   /* верхня межа */
    step = 20;     /* розмір кроку */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f %6.1f\n", fahr, celsius);
        fahr = fahr + step;

    }
}
```

Цей варіант дуже подібний до попереднього, за винятком того, що `fahr` і `celsius` оголошено як `float`, а саму формулу перетворення написано у натуральніший спосіб. Ми не могли використати 5/9 у попередній програмі, оскільки ділення цілих округлилося би до нуля. Десяткова точка у константі вказує на те, що це — числом з рухомою точкою, тож 5.0/9.0 не округляється.

Якщо арифметичний оператор має лише цілі операнди, відбудеться дія з цілими. Якщо ж арифметичний оператор має один операнд, який є числом з рухомою точкою, і один операнд-ціле, тоді ціле буде перетворено на число з рухомою точкою. Тож, якби ми написали `(fahr-32)`, 32 автоматично перетворилося б у дріб. Тим не менше, написання констант з десятковою частиною підкреслює для читачів коду той факт, що вони мають справу з числами з десятковою точкою.

Ви знайдете детальніший опис того, коли цілі перетворюються у числа з рухомою точкою у Розділі 2. Поки-що зауважте, що присвоєння

```
fahr = lower;
```

і тестування

```
while (fahr <= upper)
```

також працюють у натуральний спосіб — `int` перетворено у `float` перед тим як здійснити операцію.

Вказівник перетворення `%3.0f` функції `printf` вказує на те, що потрібно вивести число з рухомою точкою (тут `fahr`), шириною, щонайменше, три знаки, без десяткової крапки та дробової частини. `%6.1f` описує інше число (`celsius`), яке буде виведено, щонайменше, шириною шість знаків, з однією цифрою після десяткового знака. Вивід виглядатиме наступним чином:

```
0           -17.8
20          -6.7
40          4.4
...
```

Ширина і точність може бути опущеною у вказівникові: `%6f` вказує на те, що число повинно бути, принаймні, шириною шість знаків, `%.2f` вказує на два знаки після десяткової крапки, але без обмеження ширини, а `%f` — просто вивід числа з рухомою точкою.
`%d` вивести як десяткове ціле `%6d` вивести як десяткове ціле шириною, щонайменше, 6 знаків `%f` вивести як число з рухомою точкою `%6f` вивести як число з рухомою точкою шириною, щонайменше, 6 знаків `%.2f` вивести як число з рухомою точкою з двома знаками після десяткової крапки `%6.2f` вивести як число з рухомою точкою шириною, щонайменше, 6 знаків з 2-а після десяткової крапки

Серед інших, `printf` також розпізнає `%o` для вісімкового, `%x` — для шістнадцяткового, `%c` — для символу, `%s` — для символьного ланцюжка, і `%%` — для відображення самого `%`.

Вправа 1-3. Змініть програму по перетворенню температур таким чином, щоб вона виводила заголовок над таблицею.

Вправа 1-4. Напишіть програму, яка би виводила відповідну таблицю перетворень з Цельсія у Фаренгейт.

1.3 Твердження for

Існує багато способів написання програми для вирішення одного і того ж завдання. Тож, спробуймо інший варіант перетворювача температур.

```
#include <stdio.h>

/* виводить таблицю Фаренгейт-Цельсій */
main()
{
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Це спричиняє до тих самих відповідей, але без сумніву виглядає інакше. Одна з основних перемін — це видалення більшості змінних; залишилась тільки `fahr`, і ми

оголосили її як `int`. Верхня й нижня межа та крок представлені як константи у твердженні `for`, — нової для нас конструкції. Вираз, що обчислює температуру за Цельсієм, з'являється як третій аргумент `printf`, замість окремого виразу присвоєння.

Ця остання зміна є прикладом загального правила — у будь-якому контексті, де вживається значення певного типу, ви можете використати складніший вираз того самого типу. Оскільки третім аргументом `printf` має бути число з рухомою точкою, щоб зійтися з `%6.1f`, то будь-який вираз, який повертає число з рухомою точкою, може зайняти це місце.

Твердження `for` також є циклом — узагальненим випадком `while`. Якщо ви порівняєте його з попереднім `while`, то робота `for` стане зрозумілою. Всередині дужок існують три частини, розділені крапкою з комою. Перша частина, ініціалізація

```
fahr = 0
```

відбувається один раз, ще до того, як увійти до циклу. Друга частина — це перевірка умови, яка контролює цикл:

```
fahr <= 300
```

Ця умова обчислюється; якщо вона істинна, буде виконано корпус циклу (в цьому випадку один вираз `printf`). Після цього виконується стадія приросту

```
fahr = fahr + 20
```

і умова оцінюється знову. Цикл завершиться тільки тоді, коли умова виявиться хибною. Так само як і з `while`, корпус циклу може містити одне твердження або групу тверджень, включених у фігурні дужки. Ініціалізацією, умовою та приростом може служити будь-який вираз.

Вибір між `while` і `for` є довільним і може залежати тільки від того, який з них здається зрозумілішим. `for`, як правило, підходить для циклів, в яких ініціалізація та приріст складаються з одного виразу кожен, і вони логічно пов'язані між собою. Це компактніше за `while` і зберігає вирази, які контролюють цикл, разом, в одному місці.

Вправа 1-5. Змініть програму перетворення температур, щоб вона виводила таблицю у зворотній послідовності, тобто від 300 градусів до 0.

1.4 Символічні константи

Останнє спостереження, до того як ми назавжди залишимо програму перетворення температур. Вважається поганою практикою закопувати «магічні числа», такі як 300 та 20 десь всередині програми; вони мало про що кажуть тим, хто переглядає програму пізніше, і їх важко змінити у систематичний спосіб. Одним з рішень питання про

«магічні числа» є надання їм осмислених імен. Рядок `#define` означує символічну назву або символічну константу для ланцюжка якихось знаків:

```
#define назва заміна
```

Таким чином, будь-яку появу назви (не в лапках, і не як частина іншої назви) буде замінено на відповідний текст заміни. Назва має ту саму форму, що й назва змінних: послідовність літер і цифр, що починаються з літери. Текст заміни може складатися з довільної послідовності знаків; він не обмежений тільки числами.

```
#include <stdio.h>

#define LOWER       0      /* нижня межа температурної шкали */
#define UPPER       300    /* верхня межа */
#define STEP        20     /* розмір кроку */

/* виводить таблицю Фаренгейт-Цельсій */
main()
{
    int fahr;

    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));

}
```

Величини `LOWER`, `UPPER` та `STEP` — це символічні константи, а не змінні, тож вони не з'являються в оголошеннях. Назви символічних сталих традиційно пишуться великими літерами, щоб легко було відрізнити їх від назв змінних малими. Зверніть увагу, що крапки з комою немає в кінці рядка `#define`.

1.5 Ввід і вивід знаків

Ми розглянемо групу споріднених програм для опрацьовування символьних даних. Ви зрозумієте пізніше, що багато програм — це просто розширені версії прототипів, які ми обговорюватимемо тут.

Модель вводу та виводу, підтримувана стандартною бібліотекою, — досить проста. Текстовий ввід або вивід, незалежно від того звідки він походить або куди направлено, розглядається як потік знаків. Текстовий потік — це послідовність знаків, розділених на рядки, де кожний рядок складається з нуля або більше символів з наступним знаком нового рядка. Це залишається відповідальністю бібліотеки — добитися того,

щоб кожний потік вводу або виводу відповідав цій моделі. С-програміст не повинен перейматись тим, як представлені рядки поза межами програми.

Стандартна бібліотека передбачає декілька функцій читання по одному знаку за раз, з яких `getchar` і `putchar` є найпростішими. Кожний раз як її викликано, `getchar` зчитує наступний введений знак із текстового потоку та повертає цей знак як власне значення. Тобто, після

```
c = getchar();
```

змінна c міститиме наступний знак вводу. Знаки, як правило, надходять з клавіатури; ввід з файлів ми обговоримо в Розділі 7.

Функція `putchar` виводить один знак кожного разу як її викликано:

```
putchar(c);
```

виводить як знак вміст цілочисельної змінної `c`; типово вивід надходить на екран. Виклики `putchar` і `printf` можна чергувати; вивід з'являтиметься в тій послідовності, в якій здійснено виклики.

1.5.1 Копіювання файла

Маючи `getchar` і `putchar` ви можете написати напрочуд багато корисного коду, не знаючи більше нічого про ввід і вивід. Найпростіший приклад — це програма, що копіює свій ввід до власного виводу по одному знаку за раз:

```
прочитати знак
while (знак не є вказівником кінця файла)
    вивести щойно прочитаний знак
    прочитати наступний знак
```

Переклад цього у С дасть нам:

```
#include <stdio.h>

/* копіює ввід до виводу; 1-а версія */
main()
{
    int c;

    c = getchar();
    while (c != EOF) {
        putchar(c);
```

```
        c = getchar();
    }
}
```

Порівнювальний оператор `!=` означає «не дорівнює».

Те, що здається знаком на клавіатурі або екрані, звичайно як і все інше, зберігається внутрішньо, як послідовність бітів. Тип `char` спеціально призначений для зберігання таких знакових даних, хоча для цього можна використати будь-який тип цілого. Ми скористалися `int` з тонких але важливих міркувань.

Проблема полягає у відокремленні кінця вводу від чинних даних. Розв'язання її пов'язане з фактом, що `getchar` повертає відмінне значення, коли немає більше вводу — значення, яке не можна плутати з якимось дійсним знаком. Воно називається `EOF`, що походить від «end of file». Ми повинні оголосити `c` такого типу, який би був досить великим для збереження будь-якого значення, поверненого `getchar`. Ми не можемо скористатися `char`, оскільки `c` повинна бути досить місткою, щоб втримати `EOF`, окрім звичайних символів. Саме тому, ми вдалися до `int`.

`EOF` — це ціле, визначене в <stdio.h>. Його типова величина не настільки важлива, доки вона не збігається зі значенням якогось знака. Використовуючи символічну константу `EOF`, ми також переконуємося, що нічого в програмі не залежить від певного числового значення.

У досвідчених програмістів, програма копіювання виглядатиме стисліше. В мові C будь-яке присвоєння на кшталт

```
c = getchar();
```

є виразом і має значення, що дорівнюватиме значенню з лівого боку після присвоєння. Це означає, що присвоєння може з'являтися як частина більшого виразу. Якщо присвоєння символу змінній `c` помістити в тестову частину циклу `while`, програму копіювання можна написати так:

```
#include <stdio.h>

/* копіює ввід до виводу; 2-а версія */
main()
{
    int c;

    while ((c = getchar()) != EOF)
        putchar(c);
}
```

Цикл `while` добуває символ, присвоює його `c`, потім перевіряє, чи цей символ не був вказівником кінця файла. Якщо ні, виконується корпус `while`, виводячи символ. Після цього `while` повторюється. По досягненню кінця вводу, `while` завершується; так само `main`.

Ця версія централізує ввід — існує тільки одне посилання на `getchar`, і це робить програму стислішою. Отримана в результаті програма, компактніша і, як тільки ви оволодієте ідіомою, легше читається. Ви часто зустрінете цей стиль. (Існує проте небезпека захопитися, і створити непроникний для розуміння код; ми намагатимемось уникати цієї тенденції.)

Дужки навколо присвоєння всередині умови обов'язкові. `!=` має більший пріоритет за `=`, що означає, що за відсутності дужок порівнювальна перевірка `!=` відбулася би до присвоєння. Тому вираз

```
c = getchar() != EOF
```

рівнозначний

```
c = (getchar() != EOF)
```

Останнє призводить до небажаного ефекту присвоєння с значень 0 або 1, залежно від того, чи повернув виклик `getchar` кінець файла, чи ні. (Більше про це можна знайти в Розділі 2.)

Вправа 1-6. Перевірте, чи вираз `getchar() != EOF` дорівнює 0 або 1.

Вправа 1-7. Напишіть програму, яка би виводила значення `EOF`.

1.5.2 Відлік символів

Наступна програма лічить символи; вона подібна до попередньої програми копіювання.

```
#include <stdio.h>

/* лічить символи вводу; 1-а версія */
main()
{
    long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

Вираз

```
++nc;
```

знайомить нас з новим оператором, ++, який означає «збільшити на одиницю». Ви могли би натомість написати `nc = nc + 1`, але `++nc` стисліший і часто — ефективніший. Існує також відповідний оператор - для зменшення на одиницю. Оператори ++ й - можуть бути префіксними (`++nc`) та постфіксними (`nc++`); ці дві форми мають різне значення у виразах, як буде показано в Розділі 2, але `++nc` й `nc++`, обидва, збільшують `nc` на одиницю. Для наших цілей, ми зупинимось на префіксній формі.

Програма відліку знаків зберігає кількість знаків у змінній типу `long` замість `int`. Довгі цілі мають, щонайменше, 32-бітну довжину. Хоч на деяких машинах `int` та `long` однакової довжини, на інших `int` має лише 16 біт, з максимальним значенням 32767, і потрібно дуже мало вводу, щоб переповнити `int`-лічильник. Вказівник перетворення `%ld` вказує `printf`, що відповідний аргумент є довгим цілим.

Ми можемо упоратись і з більшими числами, застосувавши тип `double` (число з рухомою точкою подвійної точності). Ми також використаємо твердження `for` замість `while`, для демонстрації іншого способу написання циклу.

```
#include <stdio.h>

/* лічить символи вводу; 2-а версія */
main()
{
    double nc;

    for (nc = 0; gechar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

`printf` використовує `%f` для обох типів, `float` і `double`; `%.0f` пригнічує вивід десяткової крапки та дробової частини, яку ми вказали як нуль.

Корпус цього циклу порожній, оскільки вся робота вже зроблена у тестовій та інкрементній частині циклу. Але граматичні правила C вимагають, щоб твердження `for` мало корпус. Поодинока крапка з комою, яку називають «нульовим твердженням» знаходиться там для того, щоб задовольнити це правило. Ми розмістили її на окремому рядку, щоб її було помітно.

Перед тим як ми покинемо програму-лічильник, зверніть увагу, що якщо ввід не містить жодних знаків, тести `while` або `for` зазнають невдачі при першому ж викликові `getchar` і програма виведе нуль — правильне значення. Це важливо. Однією з гарних рис `while` чи `for` є те, що вони здійснюють перевірку на вершечку циклу, до

переходу до виконання самого корпусу. Якщо робити нічого не треба, нічого й не буде зроблено, навіть якщо це означає не входити жодного разу в корпус циклу. Програми мають поводитись розумно, коли їм надано ввід нульової довжини. Твердження `while` та `for` допомагають упевнитися, що програми здійснюють розумні речі з граничними умовами.

1.5.3 Відлік рядків

Наступна програма лічить введені рядки. Як ми згадали вище, стандартна бібліотека забезпечує тим, щоб потік ввідного тексту з'являвся як послідовність рядків, кожен з яких закінчується символом нового рядка. Тому відлік рядків — це просто підрахунок символів нового рядка:

```
#include <stdio.h>

/* лічить рядки вводу */
main()
{
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
        printf("%d\n", nl);
}
```

Корпус `while` тепер включає умову `if`, яка, в свою чергу, керує приростом `++nl`. Твердження `if` перевіряє умову в дужках, і якщо вона істинна, виконує наступне твердження (або групу тверджень у фігурних дужках). Ми знову намагались показати, що контролюється чим.

Подвійний знак рівності `==` є нотацією C для «рівний з» (подібний до одного знака `=` Pascal або `.EQ` мови Fortran). Використовуються два символи рівності `==`, щоб відрізнити перевірку на рівність від одного `=`, який у C означає присвоєння. Невеличке застереження: новачки C іноді пишуть `=` там, де вони мають на увазі `==`. Як ми побачимо з Розділу 2, результат зазвичай складає чинний вираз, тож ви не отримаєте жодного попередження.

Символ в одинарних лапках повертає ціле, рівне числовому значенню знака в наборі символів машини. Це називається символьною константою (сталою), а насправді — просто інший спосіб написання невеличкого цілого. Таким чином, наприклад, `'A'` — це символьна константа, значення якої дорівнює 65 в наборі символів ASCII, — внутрішньому представленні знака A. Звичайно `'A'` надається перевага перед `65`, оскільки значення першого очевидніше, і не залежить від певного набору знаків.

Дозволяється також використання екранованих послідовностей, як символьні константи, тож '\n' означає значення символу нового рядка, яке дорівнює 10 в ASCII. Вам слід звернути увагу на те, що '\n' — це єдиний знак і у виразах є просто цілим числом, з іншого боку, "\n" — це ланцюжкова константа, яка, так сталося, що містить тільки один знак. Тему ланцюжків у порівнянні із символами розглянуто далі у Розділі 2.

Вправа 1-8. Напишіть програму з підрахунку пробілів, табуляції та нових рядків.

Вправа 1-9. Напишіть програму, яка би копіювала свій ввід до виводу, замінюючи кожний ланцюжок з одного або більше пробілів на єдиний пробіл.

Вправа 1-10. Напишіть програму, яка би копіювала свій ввід до виводу, замінюючи кожну табуляцію на \t, кожний реверс на \b і кожну зворотню похилу на \\. Це зробить табуляцію і реверси видимими у недвозначний спосіб.

1.5.4 Відлік слів

Четверта, з нашого набору корисних програм, лічить рядки, слова та знаки, з приблизним визначенням, що слово — це будь-яка послідовність знаків, що не містить пробілів, табуляції або символу нового рядка. Це спрощена версія Unix-програми wc.

```
#include <stdio.h>

#define IN        1       /* всередині слова */
#define OUT       0       /* зовні слова */

/* лічить рядки, слова та знаки вводу */
main()
{
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;

        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c = '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
```

```
    printf("%d %d %d\n", nl, nw, nc);
}
```

Кожний раз, як програма зустрічає перший знак слова, вона додає до рахунку ще одне слово. Змінна `state` занотовує, чи програма у дану мить знаходиться всередині слова, чи ні; початково вона не «у слові», маючи значення `OUT`. Ми надаємо перевагу символічним константам `IN` і `OUT` перед буквальними значеннями 1 і 0, оскільки перші роблять програму зрозумілішою. Якщо це маленька програма як от ця, різниця не відчутна, але в більших програмах, покращення прочитності варте цього невеличкого зусилля — написати саме так з самого початку. Ви також дійдете висновку, що набагато легше впроваджувати широкі зміни в програмах, де «магічні» числа з'являються тільки як символічні константи.

Рядок

```
nl = nw = nc = 0;
```

встановлює всі три змінні у значення нуль. Це не є спеціальним випадком — скоріше наслідок того, що присвоєння є виразом із певним значенням, і присвоєння спрягаються справа наліво. Це так, ніби ми написали б

```
nl = (nw = (nc = 0));
```

Оператор || означає АБО, тож рядок

```
if (c == ' ' || c == '\n' || c = '\t')
```

можна озвучити як «якщо `c` є пробілом АБО `c` є символом нового рядка АБО `c` є кроком табуляції». (Якщо пригадуєте, екранована послідовність `\t` є видимим представленням табуляції.) Існує відповідний оператор `&&`, що означає логічне І (ТА), його пріоритет вищий за ||. Вирази, поєднані `&&` або || оцінюються зліва направо і гарантовано, що оцінювання припиниться, як тільки істинність чи хибність стане відомою. Якщо `c` є пробілом, то необхідність перевіряти, чи `c` дорівнює символу нового рядка, чи табуляції відпадає, тож ці перевірки опущено. Це не настільки важливо тут, але суттєво в складніших випадках, як ми скоро побачимо.

Цей приклад також демонструє `else`, який описує альтернативну дію, якщо умовна частина твердження `if` виявиться хибною. Загальною формою є

```
if (вираз)
    твердження1
else
    твердження2
```

Одне, і тільки одне, з двох тверджень, пов'язаних з `if-else`, буде виконано. Якщо вираз в дужках є істинним, буде виконано *твердження*$_1$, якщо ні — *твердження*$_2$. Твердження можуть бути як одним, так і багатьма, включеними у фігурні дужки. У програмі підрахунку слів, після `else` знаходиться `if` з двома твердженнями, включеними у фігурні дужки.

Вправа 1-11. Як би ви перевірили програму підрахунку слів? Які типи вводу ймовірно виявлять помилки, якщо такі є?

Вправа 1-12. Напишіть програму, яка би виводила свій ввід по одному слову на рядок.

1.6 Масиви

Тепер напишімо програму, яка підрахує кількість кожної цифри, пропусків (пробіл, табуляція і знак нового рядка) і решти знаків. Це трохи штучно, але дозволяє проілюструвати декілька аспектів C в одній програмі.

У нас є дванадцять категорій можливого вводу, тож має зміст використати масив для утримування числа повторень тієї самої цифри, замість десяти окремих змінних. Ось одна з можливих версій програми:

```
#include <stdio.h>

/* лічить цифри, пропуски та інші знаки */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

     printf("digits =");
     for (i = 0; i < 10; ++i)
         printf(" %d", ndigit[i]);
     printf(", white space = %d, other = %d\n", nwhite, nother);
```

```
}
```

Вивід самої програми може виглядати як

```
digits = 9 3 0 0 0 0 0 0 0 1, white space = 123, other = 345
```

Оголошення

```
int ndigit[10];
```

описує `ndigit`, як масив з 10-и цілих. Індексація масивів завжди починається з нуля в C, тож елементами будуть `ndigit`[0], `ndigit`[1], ..., `ndigit`[9]. Це відображено у циклах `for`, які ініціалізують і виводять масив.

Індексом може бути будь-який вираз типу `int`, включаючи цілочисельну змінну, як от `i`, та цілочисельні константи.

Ця програма покладається на символьне представлення цифр. Так, наприклад, перевірка

```
if (c >= '0' && c <= '9')
```

визначає, чи символ, який міститься в `c` є цифрою. Якщо так, числовим значенням цієї цифри є

```
c - '0'
```

Це працює тільки за умови, що символи `'0'`, `'1'`, ..., `'9'` мають послідовно-зростаючі значення. На щастя, це справджується в усіх наборах символів.

За означенням, `char` — це просто малі цілі, тож змінні і сталі типу `char` тотожні `int` в арифметичних виразах. Це природньо та зручно; наприклад, `c - '0'` є цілочисельним виразом зі значенням між 0 і 9, що відповідають знакам від `'0'` до `'9'`, збереженим у `c`, тож чинним індексом масиву `ndigit`.

Рішення того, чи знак є цифрою, пропуском, чи чимось іншим здійснюється послідовністю

```
if (c >= '0' && c <= '9')
    ++ndigit[c-'0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;
```

Конструкція

```
if (умова1)
    твердження1
else if (умова2)
    твердження2
    ...
    ...
else
    твердження
```

зустрічається доволі часто в програмах, як один з способів виразити розгалуження рішень. Умови розглянуто по-порядку, починаючи зверху, до тих пір, доки одна з умов не справдиться, у разі чого буде виконано відповідне твердження, і ціла конструкція закінчить своє існування. Якщо жодна з умов не є істинною, тоді буде виконано твердження після останнього `else`, якщо таке існує. Якщо ж останнє `else` і відповідне твердження відсутні, як у випадку з програмою відліку слів, тоді жодної дії не відбудеться. Можна використати будь-яку кількість

```
else if (умова)
    твердження
```

-груп, між початковим `if` і кінцевим `else`.

Щодо стилю, то радимо форматувати цю конструкцію саме так, як ми показали; якби кожний `if` вирівнювався з попереднім `else`, довга черга розгалужень змістилась би до правого боку сторінки.

Твердження `switch`, яке буде розглянуто у Розділі 4, — це інший спосіб написання розгалуження рішень, особливо корисне у випадку, коли умова складається з якогось цілого чи символьного виразу, який порівнюється з набором констант. Для контрасту, у Розділі 3.4 ми представимо `switch`-версію цієї програми.

Вправа 1-13. Напишіть програму, яка би виводила гістограму довжин слів вводу. Гістограму легко намалювати горизонтальними стрижнями; вертикальну орієнтацію гістограми втілити трохи складніше.

Вправа 1-14. Напишіть програму виводу гістограми частоти різних знаків вводу.

1.7 Функції

У C, функція — це еквівалент підпрограм чи функцій Fortran, або процедур чи функцій Pascal. Функції забезпечують зручним способом герметизувати, або відокремити, якесь обчислення, яке після того можна використати не хвилюючись про те, як саме воно було втілене. Із добре спроектованими функціями, можна не звертати уваги, як

саме вирішено проблему; знання того, що саме зроблено — вистачить. С робить використання функцій легким, зручним і ефективним; ви часто побачите короткі функції, означені та викликані тільки один раз лише тому, що вони прояснюють якийсь кусочок коду.

Досі ми використовували тільки такі функції як `printf`, `getchar` і `putchar`, які нам було надано; тепер час написати декілька власних. Оскільки С не має експоненційного оператора `**`, як у Fortran, дозвольте нам продемонструвати механіку визначення функції шляхом написання власної `power(m,n)`, яка зводить ціле `m` до додатнього чілочисельного степеню `n`. Тобто, значенням `power(2,5)` буде 32. Ця функція не є практичною рутиною зведення до степеню, оскільки вона оперує лише додатними показниками степеня з невеликими значеннями, але вона підходить для ілюстрації. (Стандартна бібліотека містить функцію `pow(x,y)`, яка обчислює x^y.)

Наступне — це функція `power` разом із `main` для її виклику, тож ви можете побачити всю структуру зразу.

```
#include <stdio.h>

int power(int m, int n);

/* випробовування функції power */
main()
{
    int i;

    for (i = 0; i < 10; ++i)
        printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}

/* power:  зводить base до n-ного степеня; n >= 0 */
int power(int base, int n)
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Визначення функції має таку форму:

```
тип_повернення назва_функції(оголошення параметрів, якщо є)
```

```
{
    оголошення

    твердження
}
```

Визначення функцій можуть з'являтися в будь-якій послідовності, в одному вихідному файлі або в багатьох, за умови, що функцію не розщеплено по різних файлах. Якщо вихідний текст програми розбито на декілька файлів, вам, можливо, доведеться здійснити додаткові дії, щоб скомпілювати та завантажити її, ніж коли все знаходиться в одному, але це залежить від операційної системи, а не властивостей мови. Наразі, ми припустимо, що обидві функції знаходяться у тому самому файлі, тож все, чого ви навчилися про запуск С-програм, працюватиме. Функцію `power` викликано двічі всередині `main` у рядкові

```
printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Кожний виклик передає два аргументи функції `power`, яка, в свою чергу, повертає ціле для форматування і виводу. Всередині виразу `power(2,i)` є цілим, так само як `2` та `i`. (Не всі функції видають ціле значення; ми розглянемо це питання у Розділі 4.)

Перший рядок самої `power`

```
int power(int base, int n)
```

оголошує типи параметрів та їхні назви, і тип результату, який функція повертає. Назви, використані `power` для свої параметрів, є локальними для `power`, і не видимі для будь-якої іншої функції — інші функції можуть скористатися з тих самих назв, не викликаючи конфліктів. Те саме стосується змінних `i` та `p` — змінна `i` з `power` не має жодного стосунку до `i` з `main`.

Ми, загалом, користуватимемося словом «параметр» для назв змінних функції зі списку в круглих дужках. Терміни «формальний аргумент» і «дійсний аргумент» також іноді використовуються для такого розрізнення.

Значення, обчислене `power`, передається `main` за допомогою твердження `return`. Будь-який вираз може слідувати за `return`:

```
return вираз;
```

Функції не обов'язково мають повертати якесь значення; твердження `return` без якогось виразу передає контроль, але жодного корисного значення, викликачеві, так само як «падіння з кінця» функції, коли досягнуто кінцевої фігурної дужки. Викликова функція також може проігнорувати значення, повернене викликаною.

Ви, можливо, помітили `return` укінці `main`. Оскільки `main` — це така сама функція, як і будь-яка інша, вона також може повертати значення викликачеві, що насправді є середовищем у якому запущено програму. Типово, повернення нуля означає нормальне завершення; ненульові значення сигналізують незвичайні або помилкові умови завершення. Дотепер, ми, для спрощення, опускали твердження return всередині main, але надалі ми включатимемо його, як нагадування, що програми мають повертати середовищу свій статус.

Оголошення

```
int power(int base, int n);
```

перед самою `main` вказує на те, що `power` — це функція, що очікує два аргументи типу `int` і повертає один `int`. Це оголошення, яке називається прототипом функції, має збігатися з визначенням і використанням `power`. Це спричинить помилку, якщо визначення функції або якийсь випадок її використання не зійдеться з прототипом.

Назви параметрів не мусять збігатися. Насправді, назви параметрів не обов'язкові в прототипі функції, тож його можна написати як

```
int power(int, int);
```

Вдало вибрані імена змінних — це хороша підказка, однак, тож ми часто їх використовуватимемо. Історична примітка: найбільшою зміною між ANSI C і ранніми версіями є спосіб в який функції оголошено і означено. В оригінальній версії C, функцію `power` було би написано так:

```
/* power: зводить base до n-ного степеня; n >= 0 (старий стиль) */
power(base, n)
int base, n;
{
    int i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

Параметри вказано всередині круглих дужок, а їхні типи — перед відкриттям фігурних; неоголошені параметри вважаються за `int`. (Корпус функції такий самий як і у попередньому прикладі.)

Оголошення `power` на початку програми виглядало би так:

```
int power();
```

Список параметрів був неможливим, тож компілятор не міг одразу перевірити, чи `power` буде викликано належним чином. Насправді, оскільки `power` без задання всерівно повернуло би `int`, оголошення можна було би опустити взагалі.

Новий синтаксис прототипів функцій набагато полегшує компілятору виявлення помилок у кількості аргументів або їхніх типах. Старий стиль оголошень і визначень ще працює в ANSI C, принаймні протягом перехідного періоду, але ми настійно рекомендуємо, щоб ви вживали нову форму, якщо ваш компілятор її підтримує.

Вправа 1-15. Перепишіть заново програму перетворення температур з Розділу 1.2 з використанням функції для перетворень.

1.8 Аргументи - виклик за значенням

Одна з рис функцій C можливо не знайома програмістам, які звикли до інших мов, зокрема Fortran. У C, аргументи функцій передаються «за значенням». Це означає, що викликаній функції передаються значення аргументів у вигляді тимчасових змінних, а не оригінали. Це призводить до дещо відмінних властивостей, ніж ті, які можна зустріти у разі «виклику за зверненням», типового для Fortran, або у випадку параметрів var у Pascal, в якому викликана функція має доступ до оригінального аргументу, а не локальної копії.

Проте, виклик за значенням — це перевага, а не перешкода. Він, як правило, веде до компактніших програм з меншою кількістю сторонніх змінних, оскільки з параметрами можна обходитись як зі зручно ініціалізованими локальними змінними у викликаній функції. Ось, наприклад, версія `power`, яка користується цією властивістю.

```
/* power:  зводить base до n-ного степеня; n >= 0 */
int power(int base, int n)
{
    int p;

    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

Параметр `n` використовується як тимчасова змінна, значення якої поступово зменшується (цикл `for`, у оберненому напрямку), доки `n` не дорівнюватиме нулю. Потреба у змінній `i` відпадає. Що би не відбулося з `n` всередині `power`, це не матиме жодного впливу на аргумент, з яким `power` було початково викликано.

Якщо треба, можна також примусити функцію змінити змінну у виликовій функції. В такому разі, викликач має вказати адресу змінної, над якою відбудеться дія (тобто, покажчик на змінну), а викликана функція повинна оголосити параметр, як покажчик, і звернутися до змінної непрямо через нього. Ми розглянемо покажчики в Розділі 5.

З масивами — інша історія. Коли як аргумент вказано назву масиву, тоді значення передане функції складатиметься з місцезнаходження, тобто адреси початку масиву; жодного копіювання елементів масиву не відбувається. Через індексацію цього значення, функція може звернутися і змінити будь-який елемент масиву. Саме це буде темою наступного розділу.

1.9 Символьні масиви

Найпоширенішим типом масивів у C є масиви символів. Щоб проілюструвати використання символьних масивів і функцій для їхньої обробки, давайте напишемо програму, яка читає набір рядків тексту і друкує найдовший. Схема є доволі простою:

```
while (є ще один рядок)
    if (він довший за попередній найдовший)
        (зберегти його)
        (зберегти його довжину)
вивести найдовший рядок
```

Ця схема робить зрозумілим, що програма природньо поділяється на частини. Одна частина добуває новий рядок, інша зберігає його, а решта керує процесом.

Оскільки речі можна так гарно розділити, було би непогано написати їх саме таким чином. Отже, напишімо спочатку окрему функцію `getline`, яка добуватме наступний рядок вводу. Ми спробуємо зробимо цю функцію корисною і в інших контекстах. Щонайменше, `getline` повинна сигналізувати про можливий кінець вводу; краща конструкція повертала би довжину рядка або нуль, якщо досягнуто кінця вводу. Нуль є прийнятним вказівником кінця вводу, оскільки це ніколи не може бути чинною довжиною рядка. Будь-який рядок тексту має, принаймні, один знак, навіть рядок, що містить тільки символ нового рядка, матиме довжину 1.

Коли ми знайдемо рядок, довший за попередній найдовший, його треба буде десь зберегти. Це наводить на думку про другу функцію, `copy`, яка копіюватиме новий рядок у надійне місце.

І, нарешті, нам потрібна програма `main`, для керування `getline` і `copy`. Ось, що ми отримали в результаті.

```
#include <stdio.h>
#define MAXLINE 1000        /* максимальна довжина рядка вводу */
```

```
int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* виводить найдовший рядок вводу */
main()
{
    int len;                /* довжина поточного рядка */
    int max;                /* найбільша знайдена довжина */
    char line[MAXLINE];     /* поточний рядок вводу */
    char longest[MAXLINE];  /* найдовший рядок буде збережено тут */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }
    if (max > 0)            /* було знайдено рядок */
        printf("%s", longest);
    return 0;
}

/* getline: зчитує рядок у s, повертає довжину */
int getline(char s[],int lim)
{
    int c, i;

    for (i=0; i < lim-1 && (c=getchar())!=EOF && c!='\n'; ++i)
        s[i] = c;
    if (c == '\n') {
        s[i] = c;
        ++i;
    }
    s[i] = '\0';
    return i;
}

/* copy: копіює from до to; припускає, що to досить великий */
void copy(char to[], char from[])
{
    int i;

    i = 0;
```

```
    while ((to[i] = from[i]) != '\0')
        ++i;
}
```

Функції `getline` і `copy` оголошено на самому початку програми, яку, як ми припускаємо, розміщено в одному файлі.

`main` і `getline` комунікують через пару аргументів і повернене значення. В `getline`, аргументи оголошено рядком

```
int getline(char s[], int lim);
```

який вказує на те, що перший аргумент `s` є масивом, а другий, `lim` — цілим числом. Зміст вказівки розміру масиву під час оголошення полягає у тому, що це відводить місце для зберігання. Довжину масиву `s` не обов'язково вказувати в `getline`, оскільки розмір вже задано в `main`. `getline` використовує `return`, щоб передати значення назад викликачеві, так само як ми це бачили у функції `power`. У цьому рядку також зазначено, що `getline` повертає `int`; оскільки `int` — це стандартний тип повернення, його можна було би опустити.

Деякі функції повертають корисні значення — інші, такі як `copy`, використовуються тільки заради їхнього ефекту і не повертають жодних значень. Типом повернення `copy` є `void`, що явно вказує на те, що ніякого значення не повертається.

`getline` додає символ `'\0'` (нульовий символ, чиє ASCII-значення дорівнює нулю) вкінці створеного нею масиву, для позначення кінця символьного ланцюжка. Це перетворення так само застосовується в C. Коли ланцюжкова стала на зразок

```
"hello\n"
```

з'являється в C-програмі, її збережено як символьний масив, що містить знаки ланцюжка і закінчується символом `'\0'`, щоб позначити кінець ланцюжка.

h	e	l	l	o	\n	\0

Вказівник формату `%s` функції `printf` очікує, що відповідний аргумент буде ланцюжком, представлений саме в цій формі. `copy` також покладається на той факт, що її аргумент вводу закінчується `'\0'`, і копіює цей символ до виводу.

Мимохідь варто зазначити, що навіть така маленька програма як ця представляє деякі складні проблеми розробки. Наприклад, що повинна зробити `main` у випадку, коли вона зустріне рядок, більший за встановлене обмеження? `getline` діє безпечно, через те, що вона перестає набирати знаки, коли масив повний, навіть якщо не було

знака нового рядка. Перевіряючи довжину й останній повернений знак, `main` може визначити, чи не був рядок занадто довгим, після чого обійтися з ним, як їй заманеться. Заради стислості, ми проігнорували це питання.

Користувач `getline` ніяк не може знати наперед, яким за довгим буде введений рядок, тож getline перевіряє на предмет переповнення. На противагу, користувач copy вже знає (або може дізнатися) довжину ланцюжків, тож ми вирішили перевірку на помилки до неї не додавати.

Вправа 1-16. Перероблть функцію `main` програми знаходження найдовшого рядка, щоб вона правильно друкувала довжину рядків вводу довільного розміру і, наскільки це можливо, цілого тексту.

Вправа 1-17. Напишіть програму виводу усіх введених рядків довжиною понад 80 знаків.

Вправа 1-18. Напишіть програму, яка би вилучала кінцеві пробіли і табуляцію з кожного рядка вводу й усувала повністю порожні рядки.

Вправа 1-19. Напишіть функцію `reverse(s)`, яка б обертала символьний ланцюжок `s` задом на перед. Використайте її для створення програми для обертання свого вводу по одному рядкові за раз.

1.10 Зовнішні змінні й область дії

Змінні в `main`, такі як `line`, `longest` тощо, є приватними або локальними для `main`. Оскільки їх оголошено всередині `main`, жодна інша функція не може мати безпосереднього доступу до них. Те саме стосується змінних в інших функціях; наприклад, змінна `i` функції `getline` не має жодного стосунку до `i` з `copy`. Кожна локальна змінна у функції починає існувати тільки тоді, коли викликано функцію і зникає, коли функція закінчила своє виконання. Ось чому такі змінні часто називають автоматичними змінними, наслідуючи термінологію інших мов. Ми теж надалі використовуватимемо цей термін стосовно локальних змінних. (У Розділі 4 обговорюється клас зберігання `static`, в якому локальні змінні утримують своє значення між викликами функцій.)

Оскільки автоматичні змінні з'являються і зникають із викликами функцій, вони не зберігають свого значення від одного виклику до іншого, і їм щоразу треба явно присвоїти значення, інакше вони міститимуть непотріб.

Як альтернатива автоматичним, можна означити змінні, які будуть зовнішніми для всіх функцій, тобто змінні, на які будь-яка функція зможе послатися за їхнім іменем. (Цей механізм схожий на змінні `COMMON` мови Fortran або змінні, оголошені у зовнішньому блоці в Pascal.) Внаслідок того, що зовнішні змінні глобально доступні, їх можна використати замість списку аргументів для обміну даними між функціями. Більше того, завдяки тому, що зовнішні змінні існують постійно, замість того, щоб з'являтися і зникати з викликами функцій, вони зберігають своє значення навіть після того, як функція, яка їх встановила, завершила свою роботу.

Зовнішню змінну треба означити один тільки раз за межами будь-якої функції; це відведе місце для їхнього зберігання. Змінну також потрібно оголосити у кожній

функції, яка хоче мати доступ до неї; це заявить про тип змінної. Оголошення може бути явним твердженням `extern`, або неявним через контекст. Щоб конкретизувати нашу дискусію, перепишімо програму виявлення найдовшого рядка, із `line`, `longest` і `max`, як зовнішні змінні. Це вимагає зміну викликів, оголошень і тіл усіх трьох функцій.

```
#include <stdio.h>

#define MAXLINE 1000     /* максимальна довжина рядка вводу */

int max;                 /* найбільша знайдена довжина */
char line[MAXLINE];      /* поточний введений рядок */
char longest[MAXLINE];   /* зберігає найдовший  рядок */

int getline(void);
void copy(void);

/* виводить найдовший рядок вводу; спеціальна версія */
main()
{
    int len;
    extern int max;
    extern char longest[];

    max = 0;
    while ((len = getline()) > 0)
        if (len > max) {
            max = len;
            copy();
        }
    if (max > 0)      /* було введено рядок */
        printf("%s", longest);
    return 0;
}

/* getline:  спеціальна версія */
int getline(void)
{
    int c, i;
    extern char line[];

    for (i = 0; i < MAXLINE - 1
           && (c=getchar)) != EOF && c != '\n'; ++i)
               line[i] = c;
```

```
        if (c == '\n') {
            line[i] = c;
            ++i;
        }
        line[i] = '\0';
        return i;
    }
    /* copy:  спеціальна версія */
    void copy(void)
    {
        int i;
        extern char line[], longest[];

        i = 0;
        while ((longest[i] = line[i]) != '\0')
            ++i;
    }
```

Зовнішні змінні в `main`, `getline` та `copy` означено в перших рядках у прикладі вище, що вказує на їхній тип і відводить місце для зберігання. Синтаксично, зовнішні означення схожі на означення локальних змінних, але оскільки це відбувається за межами функцій, змінні стають зовнішніми. Перед тим, як якась функція може користуватися зовнішньою змінною, назва змінної повинна стати відомою функції; оголошення будуть таким самим як дотепер, за винятком доданого слова `extern`.

В деяких випадках, оголошення `extern` можна опустити. Якщо означення зовнішньої змінної відбувається у вихідному файлі перед її використанням в якійсь функції, тоді відпадає потреба в оголошенні extern всередині функції. Таким чином, оголошення `extern` у `main`, `getline` і `copy` зайві. В дійсності, загальною практикою є розміщення всіх означень зовнішніх змінних на початку вихідного файла, уникаючи таким чином оголошень `extern`.

Якщо програма розміщена в декількох вихідних файлах, і якусь змінну, означену у *файлі1*, було використано у *файлі2* і *файлі3*, тоді оголошення `extern` — обов'язкові у *файлі2* та *файлі3* для поєднання використань змінної. Звичною практикою є зібрати всі оголошення `extern` змінних і функцій в окремий файл, історично названий файлом заголовка, оскільки його включено директивою `#include` в заголовок кожного вихідного файла. Традиційним суфіксом назв файлів заголовка є `.h`. Функції стандартної бібліотеки, наприклад, оголошено у файлі заголовка на зразок <`stdio.h`>. Цю тему розглянуто в повному обсязі у Розділі 4, а саму бібліотеку — у Розділі 7 і Додатку Б.

Через те, що спеціалізовані версії `getline` та `copy` не мають аргументів, логіка підказує, що їхні прототипи напочатку файла повинні виглядати як `getline()` та `copy()`. Але заради сумісності зі старшими C-програмами, стандарт розглядає поро-

жній список аргументів, як оголошення старого стилю, і вимикає будь-яку перевірку списку аргументів; треба використати слово `void` для відверто порожнього списку аргументів. Ми обговоримо це далі у Розділі 4.

Ви, напевне, помітили, що в цьому розділі ми обережно використовуємо слова «означення» та «оголошення», коли ми посилаємося на зовнішні змінні. «Означення» стосується того місця, де змінну створено або призначено місце для зберігання; «оголошення» вживається для тих місць, де висловлено сутність змінної, але не відведено місця для її збереження.

Між іншим, багато хто схильний робити зі всього зовнішню змінну, оскільки здається, ніби це спрощує комунікацію між функціями — списки аргументів коротшають і змінні завжди під рукою, коли вони вам потрібні. Але зовнішні змінні завжди наявні, навіть тоді, коли ви їх не хочете. Занадто покладатися на зовнішні змінні — це згубний шлях, оскільки це призводить до програм, чиї сполучення даних не настільки явні — змінні можуть мінятися непередбачувано чи ненавмисно, і програму стає важко модифікувати. Друга версія програми виявлення найдовшого рядка гірша за першу частково саме з цих міркувань і частково тому, що вона зруйнувала загальність двох корисних функцій через внесення в них назв змінних, якими функції маніпулюють.

Досі, ми охопили те, що можна назвати традиційним осердям мови програмування C. З цим набором складових можна писати корисні програми значного розміру, і було би непогано, якби ви призупинилися на деякий час, щоб зайнятися цим. Наступні вправи пропонують дещо складніші програми, ніж ті, які ми бачили в цьому розділі.

Вправа 1-20. Напишіть програму `detab`, яка би замінювала табуляцію у вводі на відповідну кількість пробілів. Припустіть сталий крок табуляції, скажімо кожний n-нний стовпчик. Чи має бути n змінною, чи символічною константою?

Вправа 1-22. Напишіть програму для завертання довгих рядків вводу у два або більше коротших після останнього знака, що не є пробілом, який знаходиться перед n-нним стовпчиком вводу. Впевніться, що ваша програма діє розумно з дуже довгими рядками і у випадку браку пробілів і табуляції перед вказаним стовпчиком.

Вправа 1-23. Напишіть програму, яка би вилучала всі коментарі з вихідного файла C. Не забувайте про видалення залапкованих ланцюжків і символьних констант. Коментарі в C не гніздуються.

Вправа 1-24. Напишіть програму, яка би перевіряла C-програми на елементарні синтаксичні помилки, як от невідповідність круглих, фігурних і квадратних дужок. Не забувайте про одинарні та подвійні лапки, екрановані послідовності, коментарі. (Ця програма важка, якщо ви зробите її загальною.)

Розділ 2

Типи, оператори та вирази

Змінні та сталі — це основні об'єкти даних, якими орудує програма. Оголошення укладають список змінних, що використовуватимуться, і зазначають їхній тип і, можливо, початкове значення. Оператори вказують на виконувану дію. Вирази об'єднують змінні та сталі для утворення нового значення. Тип певного об'єкту обумовлює набір значень, який той може мати, а також, які саме операції застосовні щодо нього. Ці складові частини і є темою даного розділу.

Стандарт ANSI привніс багато малих змін і доповнень до основних типів і виразів. Тепер існують знакові (`signed`) та беззнакові (`unsigned`) форми всіх типів цілих, а також позначення для беззнакових констант і шістнадцяткових символьних констант. Операції з рухомою точкою можливі з одинарною точністю; існує також тип `long double` (довге подвійне число) для підвищеної точності. Ланцюжкові константи може бути зчеплено під час компіляції. Переліки (енумерація) стали офіційною частиною мови, формалізуючи цю довготривалу рису. Об'єкти можна оголосити як `const`, що запобігає їхній зміні. Доповнено правила автоматичного коригування серед арифметичних типів, для можливості оперування багатшим набором.

2.1 Назви змінних

Хоч ми не згадали про це в Розділі 1, існують певні обмеження щодо назв змінних та символічних констант. Назви повинні складатися з літер і цифр, першим знаком має бути літера. Жорсткий пробіл «_» теж вважається літерою, він часом корисний для покращення прочитності довгих назв змінних. Однак, не починайте назв змінних з жорсткого пробілу, оскільки функції бібліотеки часто використовують такі назви для власних потреб. Літери верхнього регістру та літери нижнього регістру різняться, тож `x` та `X` — це дві різні назви. Традиційною практикою в C є використання літер нижнього регістру для назв змінних і тільки верхнього для символічних констант.

Щонайменше, 31 знак внутрішнього імені є значущим. Для назв функцій і зовнішніх змінних число може виявитися меншим за 31, оскільки зовнішні назви можуть використовуватись асемблерами та завантажувачами зв'язків, над якими мова не має

жодного впливу. Для зовнішніх імен, стандарт гарантує унікальність тільки для 6-ох знаків одного регістру. Ключові слова, такі як `if`, `else`, `int`, `float` тощо, зарезервовано — ви не можете використати їх для назв змінних. Ключові слова мають бути нижнього регістру.

Розумним буде обирати такі назви змінних, які би відображали зміст змінної і які неможливо би було типографічно сплутати з іншими. Ми схильні вживати короткі назви для локальних змінних, особливо лічильників циклу, і довші — для зовнішніх змінних.

2.2 Типи даних і розміри

Існує лише кілька основних типів даних у C:

`char`	один байт, здатний утримувати один знак локального набору символів
`int`	ціле, типово відображає натуральний розмір цілих машини
`float`	одинарної точності число з рухомою точкою
`double`	подвійної точності число з рухомою точкою

На додачу, існує певна кількість класифікаторів, які можна використати стосовно вищевказаних основних типів. Так, `short` із `long` застосовуються з цілими:

```
short int sh;
long int counter;
```

Слово `int` можна опустити в таких оголошеннях і, типово, саме так і роблять.

Ідея полягає в тому, що `short` та `long` мають забезпечувати різними довжинами цілих, там де це має практичний сенс; `int`, як правило, буде натуральним розміром цілого для певної машини. `short` часто має 16-бітну довжину, а `int` — 16, або 32-бітну. Кожний компілятор може вибрати відповідні розміри для власного устаткування, єдиним обмеженням будучи те, що `short` та `int` повинні мати щонайменше 16 біт, `long` — 32 біти, `short` не може бути довшим за `int`, який, в свою чергу — не довшим за `long`.

Класифікатори `signed` або `unsigned` можна застосовувати щодо `char` або будь-якого цілого. Беззнакові, `unsigned`, числа завжди додатні або нуль і підлягають правилам арифметичного модуля 2^n, де n — це кількість бітів використаного типу. Тож, наприклад, якщо `char` має довжину 8 бітів, змінні типу `unsigned char` матимуть значення між 0 та 255, тоді як `signed char` — між -128 та 127 (в машині з двійковою системою). Чи звичайні `char` вживаються зі знаком, чи є беззнаковими, залежить від машини, але друковні знаки завжди додатні.

Тип `long double` вказує на число з рухомою точкою підвищеної точності. Так само, як і у випадку з цілими, розміри об'єктів з рухомою точкою залежать від реалізації; `float`, `double` і `long double` можуть представляти один, два або три відмінних розміри.

Стандартні файли заголовка <limits.h> і <float.h> містять символічні константи для всіх трьох розмірів, разом із іншими властивостями машини та компілятора. Це все розглянуто у Додатку Б.

Вправа 2-1. Напишіть програму для визначення амплітуд змінних `char`, `short`, `int` і `long`, як зі знаком, `singned`, так і беззнакових, `unsigned`, шляхом виводу відповідних значень з файлів заголовка і шляхом безпосереднього обчислення. Важче, якщо ви обчислите їх. Визначте діапазон різноманітних типів з рухомою точкою.

2.3 Константи

Цілочисельна константа, наприклад `1234`, має тип `int`. Довга константа пишеться з кінцевою `l` (англійська «ел») або `L`, як от `123456789L`; цілочисельна константа, занадто велика для того, щоб уміститися в `int`, також буде розглянута як `long`. Беззнакові константи пишуться з кінцевою `u` (англійська «ю») або `U`, а суфікс `ul` або `UL` вказує на тип `unsigned long` (беззнакове довге ціле).

Константи з рухомою точкою, повинні містити або десяткову крапку, або експоненту, або обидві; їхній тип вважається `double`, хіба що вказано якийсь інший суфікс. Суфікс `f` або `F` позначає константу з рухомою точкою, `float`; `l` або `L` вказують на `long double` (довге подвійне число з рухомою точкою).

Значення цілого можна вказати як вісімкове або шістнадцяткове, замість десяткового. Передній 0 (нуль), у випадку цілої константи, означає вісімкове число; `0x` або `0X` попереду означають шістнадцяткове. Так, наприклад, десяткове 31 може бути записано як `037` у вісімковій системі, і `0x1f` або `0x1F` — у шістнадцятковій. За вісімковими та шістнадцятковими константами може також слідувати `L`, щоб поміняти їхній тип на `long`, або `U`, щоб перетворити їх на беззнакові; `0XFUL` — це константа, яка має тип `unsigned long` (беззнакова довга) із десятковим значенням 15.

Символьна константа — це ціле, записане як символ в одинарних лапках, як от `'x'`. Значення символьної константи дорівнює числовому значенню знака в наборі знаків машини. Наприклад, у наборі знаків ASCII, символьна константа `'0'` має значення 48, що не має жодного стосунку до числового значення 0. Якщо ми напишемо `'0'`, замість числового значення 48, яке залежить від набору знаків, програму, що не залежатиме від певного числового значення, буде водночас легше читати. Символьні константи можуть брати участь у числових операціях так само, як і інші цілі, хоч їх частіше використовують для порівнянь з іншими знаками.

Деякі знаки в символьних і ланцюжкових константах можна представити як екрановані послідовності, наприклад `\n` (знак нового рядка); ці послідовності виглядають як два знаки, але означають тільки один. На додаток, довільний ряд бітів розміром один байт можна вказати як

```
'\ooo'
```

де *ooo* — це одна до трьох вісімкових цифр (0...7), або як

```
'\xhh'
```

де *hh* — це одна або більше шістнадцяткових цифр (`0...9`, `a...f`, `A...F`). Таким чином, ми можемо написати

```
#define VTAB '\013'       /* вертикальна табуляція в ASCII */
#define BELL '\007'       /* символ дзвоника в ASCII */
```

або в шістнадцятковій формі

```
#define VTAB '\xb'        /* вертикальна табуляція в ASCII */
#define BELL '\x7'        /* символ дзвоника в ASCII */
```

Ось повний набір екранованих послідовностей: \a символ сигналу (дзвоника) \\ зворотня похила \b реверс (крок назад) \? знак питання \f зміна сторінки \' одинарні лапки \n новий рядок \" подвійні лапки \r повернення каретки \ *ooo* вісімкове число \t горизонтальна табуляція \x *hh* шістнадцяткове число \v вертикальна табуляція

Символьна константа `'\0'` позначає знак зі значенням нуль, нульовий знак. `'\0'` часто використовується замість `0`, щоб підкреслити символьне походження деяких виразів, але числове значення — це просто 0.

Сталий (константний) вираз — це такий, що включає тільки сталі (константи). Такі вирази оцінюються під час компіляції, а не під час обігу програми і, відповідно, можуть використовуватись у будь-якому місці, де може стояти константа, скажімо

```
#define MAXLINE 1000
char line[MAXLINE+1];
```

або

```
#define LEAP 1  /* високосний рік */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Ланцюжкова константа, або *ланцюжковий літерал* — це послідовність із нуля або більше знаків, оточених подвійними лапками, як, наприклад

```
"I am a string"
```

або

```
"" /* порожній ланцюжок */
```

Лапки не є частиною ланцюжка, а служать лише для його обмеження. Ті самі екрановані послідовності, які використовуються в символьних константах, можна так само застосувати в ланцюжках; \" представляє знак подвійних лапок. Ланцюжкові константи можна зчепити під час компіляції:

```
"hello, " "world"
```

тотожно

```
"hello, world"
```

Це зручно для поділу довгих ланцюжків на декілька рядків вихідного тексту.

Технічно, ланцюжкова константа — це масив знаків. Внутрішнє представлення ланцюжка включає нульовий знак '\0' укінці, тож фізичне місце зберігання вимагає на один більше символів, ніж ті, що знаходяться в подвійних лапках. Цей спосіб подання означає, що не існує обмеження довжини ланцюжка, але програми повністю має просканувати ланцюжок, щоб визначити його довжину. Функція стандартної бібліотеки `strlen(s)` повертає довжину свого аргументу, символьного ланцюжка `s`, за виключенням кінцевого '\0'. Ось наша версія цієї функції:

```
/* strlen: повертає довжину s */

int strlen(char s[])
{
    int i;

    while (s[i] != '\0')
        ++i;
    return i;
}
```

`strlen` та інші ланцюжкові функції оголошено у стандартному файлі заголовка <string.h>.

Будьте обережні, щоб зуміти відрізнити символьну константу від ланцюжка, який містить один знак: 'x' — це не те саме, що «x». Перше — це ціле число, яке використовується для здобуття числового значення літери x у машинному наборі символів. Друге — масив символів, який містить єдиний знак (літеру x) і '\0'.

Існує ще один вид сталих — константи переліку. Перелік — це список сталих цілих значень, наприклад

```
enum boolean { NO, YES };
```

Перша назва типу `enum` набере значення 0, наступна — 1, і так далі, хіба що було задано явні значення. Якщо не всі значення вказано явно, тоді ті, що не задано, продовжуватимуть прогресію від останнього заданого, як показано в наступних прикладах:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
               NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
              /* FEB = 2, MAR = 3, etc. */
```

Назви в різних переліках мають відрізнятися. Значення не обов'язково повинні відрізнятися в тому самому переліку.

Переліки забезпечують зручним способом пов'язування сталих значень із назвами, як альтернатива `#define` за винятком того, що значення може бути автоматично згенеровано для вас. Навіть якщо змінні типу `enum` оголошено, компілятори не зобов'язані перевіряти, що те, що ви зберегли в такій змінній є чинним значенням для переліку. Не зважаючи на це, змінні переліку забезпечують можливістю перевірки, тож часто це краще ніж `#define`. На додаток, налагоджувач може вивести значення змінних переліку в їхній символічній формі.

2.4 Оголошення

Усі змінні потрібно оголосити до їхнього використання, хоч деякі оголошення можна зробити непрямо, через зміст. Оголошення вказує тип і містить список з однієї або більше змінних цього типу, як, наприклад

```
int  lower, upper, step;
char c, line[1000];
```

Змінні можна розподілити поміж оголошень у будь-який спосіб; списки вище так само можна було б написати як

```
int       lower;
int       upper;
int       step;
char      c;
char      line[1000];
```

Остання форма забирає більше місця, але вигідна тим, що дозволяє додати коментар до кожного оголошення, для пізніших змін.

Змінну можна також ініціювати (надати їй початкового значення) під час її оголошення. Якщо за назвою слідує знак рівності та вираз, то цей вираз служитиме ініціалізатором, як, скажімо:

```
char     esc = '\\';
int      i = 0;
int      limit = MAXLINE+1;
float    eps = 1.0e-5;
```

Якщо змінна не є автоматичною, ініціалізація відбудеться тільки один раз, за задумом — до того як програма почне своє виконання, але ініціалізатор має бути сталим виразом. Явно ініційована автоматична змінна, започатковується кожного разу при входженні у функцію або блок коду, де вона знаходиться; ініціалізатор може складатися з будь-якого виразу. Зовнішні та статичні змінні, поза вибором, ініціалізуються до нуля. Автоматичні змінні, які не мають явного ініціалізатора, отримають невизначене значення (тобто непотріб).

Для вказівки того, що значення змінної не змінюватиметься, до оголошення можна додати класифікатор `const`. У випадку масивів класифікатор `const` вказує на те, що елементи масиву залишатимуться незмінними.

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";
```

Оголошення з `const` можуть вживатися також з масивами в якості аргументів, щоб вказати на те, що функція не змінює масиву, як наприклад:

```
int strlen(const char[]);
```

Якщо відбудеться спроба змінити значення змінної типу `const`, результат такої дії залежатиме від реалізації.

2.5 Арифметичні операції

Арифметичними операторами з двома операндами є `+`, `-`, `*`, `/` і оператор коефіцієнту `%`. Поділ цілих відкидає дробову частину. Вираз

```
x % y
```

повертає залишок поділу `x` на `y`, і нуль, якщо ділення відбудеться без залишку. Наприклад, рік буде високосним, якщо він кратний 4, але не 100, за винятком того, що роки кратні 400 — теж високосні. Таким чином

```
if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0)
    printf("%d is a leap year\n", year);
else
    printf("%d is not a leap year\n", year);
```

Оператор `%` неможливо застосувати із `float` та `double`. Напрямок округлення у випадку `/` і знак результату `%` є машинозалежним для від'ємних чисел, так само як дії у випадку втрати значності або переповнення. Оператори `+` та `-` рівні за пріоритетом, але мають менший пріоритет ніж `*`, `/` і `%`, які в свою чергу, поступаються унарним `+` та `-`. Арифметичні оператори діють зліва направо.

Таблиця 2.1 у кінці цього розділу підводить підсумок пріоритетів і асоціативності (спрягання) всіх операторів.

2.6 Реляційні та логічні оператори

Реляційними операторами є

```
>    >=    <    <=
```

Вони всі мають однаковий пріоритет. Трохи нижче за пріоритетом знаходяться оператори рівності:

```
==        !=
```

Реляційні оператори наділені меншим пріоритетом за арифметичні, тож вираз на зразок `i < lim-1` розглядатиметься як `i < (lim-1)`, як і очікувалось.

Цікавішими є логічні оператори `&&` та ||. Вирази, поєднані `&&` або ||, оцінюються зліва направо, і їхня оцінка закінчується, як тільки виявлено істинність чи хибність результату. Більшість програм C покладаються на ці властивості. Ось, наприклад, цикл функції вводу, яку ми написали у Розділі 1:

```
for (i=0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Перед тим як прочитати новий знак, необхідно перевірити, чи є місце для його збереження в масиві `s`, тож треба спочатку дізнатися, чи `i < lim-1`. Якщо ця перевірка зазнає невдачі, нам не слід продовжувати далі, і читати наступний знак.

Так само, було би недоцільно перевіряти `c` на предмет `EOF` (кінця файла), якщо до цього не викликано `getchar`; саме тому виклик і присвоєння мають відбутися до перевірки символу, збереженого в `c`.

Пріоритет `&&` є більшим за ||, але обидва поступаються релятивним операторам і операторам рівності, тож вирази на зразок

```
i < lim-1 && (c=getchar()) != '\n' && c != EOF
```

не вимагають додаткових дужок. Але оскільки пріоритет `!=` переважає оператор присвоєння `=`, дужки обов'язкові у випадку

```
(c=getchar()) != '\n'
```

для того, щоб добитися бажаного результату — спочатку, присвоєння значення `c`, а потім, порівняння його з `'\n'`.

За визначенням, числове значення релятивного або логічного виразу дорівнює 1, якщо співвідношення істинне, і 0 — якщо хибне.

Унарний оператор заперечення `!` перетворює ненульовий операнд на 0 (нуль) і, навпаки, нульовий операнд на 1 (одиницю). Поширеним є застосуванням `!` в конструкціях на зразок

```
if (!valid)
```

замість

```
if (valid == 0)
```

Важко узагальнити, яка з цих форм є кращою. Конструкції на кшталт `!valid` легко читаються («if not valid» — «якщо не дійсне»), але складніші вирази буває важко зрозуміти.

Вправа 2-2. Напишіть цикл, еквівалентний наведеному вище, але без використання `&&` або ||.

2.7 Перетворення типів

Коли якийсь оператор має операнди різних типів, останні перетворюються до спільного типу, згідно невеличкого набору правил. Загалом, автоматичними перетвореннями вважаються ті, що обертають «вужчий» операнд на «ширший» без втрати інформації, як, наприклад, при перетворенні цілого в число з рухомою точкою у виразі на зразок `f + i` (де `i` містить ціле значення, а `f` — число з рухомою точкою). Вирази, які не мають змісту, як скажімо використання числа типу `float` в якості індексу —

заборонені. Вирази, в яких можливо втратити інформацію, як от у випадку присвоєння довшого типу цілого коротшому, або присвоєння числа з рухомою точкою цілому, можуть викликати попередження, але не забороняються.

Тип `char` — це просто маленьке ціле, тож цей тип може вільно вживатися в арифметичних операціях. Це забезпечує суттєвою гнучкістю в деяких випадках перетворення символів. Прикладом може служити спрощена цим наївним втіленням, функція `atoi`, яка обертає ланцюжок цифр у їхній числовий еквівалент.

```
/* atoi: перетворює s на ціле */
int atoi(char s[])
{
    int i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');

    return n;
}
```

Як ми вже згадували це в Розділі 1, вираз

```
s[i] - '0'
```

повертає числове значення символу, яке буде збережено в `s[i]`, оскільки значення `'0'`, `'1'` і так далі, утворюють неперервну послідовність в порядку зростання.

Іншим прикладом перетворення `char` на `int` є функція `lower`, яка відображає знак у нижньому регістрі набору знаків ASCII. Якщо символ не є літерою верхнього регістру, `lower` повертає її без змін.

```
/* lower: переводить c у нижній регістр; тільки ASCII */
int lower(int c)
{
    if (c >= 'A' && c <= 'Z')
        return c + 'a' - 'A';
    else
        return c;
}
```

Це працює тільки з набором ASCII, оскільки відповідні літери верхнього і нижнього регістру знаходяться на сталій відстані як числові значення й алфавіт є неперервним — немає нічого окрім літер між `A` та `Z`. Це останнє правило не є дійсним

у випадку набору символів EBCDIC, тож цей код перекладав би не тільки літери у випадку EBCDIC.

Стандартний файл заголовка <ctype.h>, описаний у Додатку Б, визначає сімейство функцій, що забезпечують можливістю перевірок і перетворень, незалежних від набору символів. Так, наприклад, функція `tolower` — це машинонезалежна заміна, наведеної вище, функції `lower`. Так само, перевірку

```
c >= '0' && c <= '9'
```

можна поміняти на

```
isdigit(c)
```

З цієї миті і надалі, ми послуговуватимемося функціями з <ctype.h>.

Існує один нюанс, що стосується перетворення знаків на ціле. Мова С не уточнює, чи змінні типу `char` є знаковими чи беззнаковими величинами. Коли `char` перетворено на `int`, чи не може це видати від'ємне ціле? Відповідь відрізняється на різних машинах, відображаючи відмінності в архітектурах. На деяких машинах, `char` із крайнім лівим бітом рівним 1 буде перетворено на від'ємне ціле («знакове розширення»). На інших, `char` зведено до `int` шляхом додання нулів із лівого боку, а отже — завжди додатній.

За визначенням, мова С гарантує, що будь-який знак у машинному стандартному друковному наборі символів ніколи не буде від'ємним, а отже завжди складатиме додатньою величину у виразах. Але довільні послідовності бітів, збережені в символьних змінних, можуть виявитися від'ємними на деяких машинах і, навпаки — додатніми на інших. Заради портабельності, вказуйте `signed` (знакове) або `unsigned` (беззнакове), якщо несимвольні дані треба зберегти в змінних типу `char`.

Реляційні вирази на кшталт `i > j` і логічні вирази, поєднані `&&` або || мають за визначенням значення 1, якщо істинні і 0, якщо хибні. Таким чином, присвоєння на зразок

```
d = c >= '0' && c <= '9'
```

встановлює `d` до 1, якщо `c` є цифрою, і 0 — якщо ні. Проте, такі функції, як `isdigit`, можуть повернути будь-яке ненульове значення у випадку істини. «Істина» в тестовій частині `if`, `while`, `for` тощо означає просто «ненульове значення», тож це не грає особливої ролі.

Неявні арифметичні перетворення працюють як і очікується. Загалом, якщо оператор, такий як `+` або `*`, має два операнди (тобто, це бінарний оператор) відмінних типів, «нижчий» тип зводиться до «вищого» до того як здійснити операцію. У Розділі 6 Додатка А точно описано правила перетворень. Якщо відсутні беззнакові операнди, наступного набору правил цілком вистачить:

- Якщо тип якогось з операндів дорівнює `long double` (довгому подвійному), інший також буде зведено до `long double`.
- Інакше, якщо ти якогось з операндів дорівнює `double` (подвійному), інший також буде перетворено на `double`.
- Інакше, якщо тип якогось з операндів дорівнює `float` (числу з рухомою точкою), інший також буде перетворено на `float`.
- Інакше, перетворити `char` (знакове) та `short` (коротке) на `int` (ціле).
- Якщо ж ти якогось з операндів дорівнює `long` (довгому), інший також буде перетворено на `long`.

Зауважте, що `float` у виразах не перетворюються автоматично на `double`; це відрізняється від оригінального визначення. Загалом, математичні функцій, як ті з <`math.h`>, використовуватимуть подвійну точність. Основною причиною використання `float` є збереження пам'яті у випадку великих масивів або, рідше, збереження часу на машинах, де арифметика з подвійною точністю — особливо ресурсоємка.

Правила перетворень ускладнюються, коли задіяні беззнакові операнди. Проблема полягає в тому, що порівнювання між знаковими і беззнаковими значеннями залежать від машини, оскільки вони покладаються на розміри різноманітних типів цілих. Для прикладу, скажімо, що `int` дорівнює 16-и бітам, а `long` — 32-ом бітам. У такому випадку, `-1L` < `1U`, оскільки `1U`, яке є беззнаковим цілим (`unsigned int`) зведено до знакового довгого (`signed long`). Але `-1L` > `1UL`, оскільки `-1L` зведено до беззнакового довгого (`unsigned long`) і, таким чином, здається великим додатнім числом.

Перетворення мають місце також під час присвоєнь; значення з правого боку зводиться до типу лівого, що й буде типом результату.

Символ перетворюється на ціле або шляхом додання знака, або ні, як буде описано нижче.

Довші цілі перетворюються на коротші або `char` шляхом відкидання зайвих бітів старшого розряду. Таким чином, при

```
int  i;
char c;

i = c;
c = i;
```

значення `c` залишиться незмінним. Це залишатиметься істиною, незалежно від того, чи використовується знакове розширення. Однак, при протилежному напрямку присвоєння, можлива втрата інформації.

Якщо `x` є типу `float`, а `i` є `int`, тоді `x = i` та `i = x`, обидва, призводять до перетворення; обернення `float` на `int` призводить до відкидання дробової частини. Коли

`double` (подвійне) перетворено на `float` (число з рухомою точкою), округлення чи відкидання дробової частини залежить від реалізації.

Оскільки аргумент виклику функції також є виразом, перетворення типів так само може мати місце під час передачі аргументів функціям. За відсутності прототипів функцій, `char` і `short` стають `int`, а `float` стане `double`. Саме тому ми оголосили аргументи функції як `int` і `double`, навіть якщо функцію викликано із `char` і `float`.

І, нарешті, можна змусити явні перетворення в будь-якому виразі з допомогою унарного оператора зведення. В конструкції на зразок

```
(назва типу) вираз
```

вираз буде перетворено до вказаного типу, згідно правил перетворення, наведених вище. Саме поняття зведення можна порівняти із тим, ніби вираз було присвоєно змінній вказаного типу, яка потім використовується замість цілої конструкції. Так, скажімо, функція бібліотеки `sqrt` очікує аргумент типу `double` (подвійного) і видасть нісенітницю, якщо ненавмисно передати щось інше. (`sqrt` оголошено в <`math.h`>.) Тож, якщо `n` є цілим, ми можемо використати

```
sqrt((double) n)
```

для перетворення значення `n` на тип `double` перед тим як передати його `sqrt`. Зауважте, що операція зведення просто видає значення відповідного типу, сама `n` залишиться незмінною. Оператор зведення має такий самий високий пріоритет як і решта унарних операторів, як буде вказано в таблиці в кінці цього розділу. Якщо аргументи оголошено через прототип функції, як і повинно відбуватися за звичайних обставин, оголошення спричинить до автоматичного зведення будь-яких аргументів під час виклику функції. Таким чином, маючи прототип функції `srqt`

```
double sqrt(double)
```

виклик

```
root2 = sqrt(2)
```

зводить ціле `2` у подвійне `2.0` без потреби явного зведення. Стандартна бібліотека включає портабельне втілення генератора псевдовипадкових чисел і функцію для ініціалізації зерна; перша ілюструє зведення:

```
unsigned long int next = 1;

/* rand: повертає псевдовипадкове ціле в межах 0..32767 */
int rand(void)
```

```
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: встановлює зерно для rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Вправа 2-3. Напишіть функцію `htoi(s)`, яка би обертала ланцюжок з шістнадцяткових цифр (включаючи можливий `0x` або `0X`) на відповідне ціле значення (int). Дозволеними цифрами є `0` до `9`, `a` до `f` або `A` до `F`.

2.8 Оператори приросту та спаду

С забезпечує двома незвичними операторами приросту та спаду змінних. Оператор приросту `++` додає 1 до свого операнду, тоді як `-`, навпаки, віднімає 1. Ми вже не раз користувалися `++` для збільшення значення змінних, як, наприклад, у

```
if (c == '\n')
    ++nl;
```

Незвична сторона полягає в тому, що як `++`, так і `-` можуть використовуватись як префіксні оператори (перед змінною, наприклад `++n`), так і постфіксні (після змінної: `n++`). В обох випадках, як наслідок — збільшується значення `n`. Але вираз `++n` збільшує `n` до того, як це значення буде використане, тоді як `n++` збільшує `n` після того, як було використане початкове значення. Це означає, що в контексті, де дійсно використовується значення, а не тільки самий ефект, `++n` і `n++` — відмінні. Якщо `n` дорівнює 5, тоді

```
x = n++;
```

присвоїть `x` значення 5, зате у випадку

```
x = ++n;
```

`x` дорівнюватиме вже 6. В обох випадках, `n` стане рівним 6. Оператори приросту та спаду можуть використовуватись тільки зі змінними; вирази на кшталт `(i+j)++` заборонені.

У контексті, коли значення не потрібне, а тільки ефект приросту, як наприклад

```
if (c == '\n')
    nl++;
```

префікс і постфікс тотожні. Але існують випадки, коли треба звернутися тільки до одного, або тільки до іншого. Наприклад, розглянемо функцію `squeeze(s,c)`, яка вилучає всі знайдені знаки `c` з ланцюжка `s`.

```
/* squeeze: вилучає всі c з s */
void squeeze(char s[], int c)
{
    int i, j;

    for (i = j = 0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
        s[j] = '\0';
}
```

Кожного разу, як знайдено не-`c`, його скопійовано до поточної позиції `j`, і тільки після цього `j` збільшено, щоб бути готовим до наступного знака. Це точний еквівалент

```
if (s[i] != c) {
    s[j] = s[i];
    j++;
}
```

Інший приклад подібної конструкції походить з функції `getline`, яку ми написали в Розділі 1, де ми можемо замінити

```
if (c == '\n') {
    s[i] = c;
    ++i;
}
```

на компактніше

```
if (c == '\n')
    s[i++] = c;
```

В якості третього прикладу, розглянемо стандартну функцію `strcat(s,t)`, яка зчеплює ланцюжок `t` із кінцем ланцюжка `s`. `strcat` припускає, що `s` має досить місця, щоб зберегти комбінацію обох ланцюжків. Так, як ми це написали, `strcat` не повертає жодного значення; версія зі стандартної бібліотеки повертає покажчик на отриманий ланцюжок.

```
/* strcat:  зчеплює t із кінцем s; s має бути досить великим */
void strcat(char s[], char t[])
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0')             /* знаходить кінець s */
        i++;
    while ((s[i++] = t[j++]) != '\0')     /* копіює t */
        ;
}
```

Одночасно з копіюванням `t` до `s` до них застосовано постфіксний `++`, щоб упевнитись, що вони в позиції для наступного проходження через цикл.

Вправа 2-4. Напишіть альтернативну версію `squeeze(s1,s2)`, яка би вилучала кожний знак із `s1`, який збігається із будь-яким знаком `s2`.

Вправа 2-5. Напишіть функцію `any(s1, s2)`, яка повертає перше положення в ланцюжку `s1` одного із знаків ланцюжка `s2`, або `-1`, якщо жодного не знайдено. (Функція `strpbrk` зі стандартної бібліотеки здійснює те саме, тільки повертає покажчик на положення.)

2.9 Розрядні оператори

Мова C забезпечує шістьма операторами для обробки бітів; їх можна застосовувати лише з цілочисельними операндами, тобто `char`, `short`, `int` і `long`, як зі знаком, так і беззнаковими. `&` розрядний I | розрядний включний АБО ^ розрядний виключний АБО << ліве зміщення >> праве зміщення ~ доповнення (унарний оператор) Розрядний I (оператор `&`) часто використовується для того, щоб приховати набір бітів, наприклад

```
n = n & 0177;
```

обнулює всі біти `n` крім молодших 7-и.

Розрядний АБО (оператор |) використовується для ввімкнення бітів:

```
x = x | SET_ON;
```

встановлює в одиницю всі біти x, що дорівнюють одному в SET_ON. Розрядний виключний АБО (оператор ^) встановлює в одиницю кожну позицію, де операнди мають відмінні біти, і в нуль там де вони збігаються.

Не слід плутати розрядні оператори & та | з логічними операторами && та ||, в яких ідеться про зліва направо оцінку істинного значення. Наприклад, якщо x дорівнює 1 а y дорівнює 2, тоді x & y оцінюється як нуль, зате x && y — як один.

Оператори зміщення << й >> здійснюють ліве та праве зміщення лівого операнда на кількість бітів, вказаних правим операндом; останній має бути додатнім числом. Таким чином, x << 2 зміщує значення x на два положення, заповнюючи вакантні біти нулями; це рівнозначно множенню на 4. Праве зміщення беззнакової величини завжди заповнюватиме звільнені біти нулями. Праве зміщення величини зі знаком заповнюватиме бітами знака («арифметичне зміщення») на деяких машинах і 0-бітами («логічне зміщення») на інших.

Унарний оператор ~ видає протилежне значення вказаного цілого; тобто, він перетворює кожний 1-біт на 0-біт, і навпаки. Наприклад

```
x = x & ~077
```

встановлює останні шість бітів x у нуль. Зауважте, що x & ~077 не залежить від довжини «слова», тому йому надається перевага над, скажімо, x & 0177700, яке припускає, що x — 16-бітна величина. Портабельна форма не забирає додаткових ресурсів оскільки ~077 — це сталий вираз, який оцінюється під час компіляції.

Як ілюстрація деяких розрядних операторів розглянемо функцію getbits(x,p,n), яка повертає (вирівняне з правого боку) n-бітне поле x, починаючи з положення p. Ми припустимо, що бітове положення 0 знаходиться з крайнього правого боку і, що n із p — це чинні додаткові значення. Так, наприклад, getbits(x,4,3) повертає три біти, що знаходяться у 4-ій, 3-ій і 2-ій позиції, вирівняні справа.

```
/* getbits: добуває n бітів, починаючи з положення p */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

Вираз x >> (p+1-n) переміщує потрібне нам поле на правий край слова. ~0 складається з одних 1-бітів; зміщення його ліворуч на n позицій за допомогою ~0<<n розмістить нулі в n бітів з правого боку; додання до цього ~ створює маску з одиниць для n бітів справа.

Вправа 2-6. Напишіть функцію setbits(x,p,n,y), яка би повертала x, в якому n бітів, починаючи з положення p, дорівнюють крайнім правим n бітам y, залишаючи решту бітів незмінними.

Вправа 2-7. Напишіть функцію `invert(x,p,n)`, яка би повертала `x`, в якому порядок `n` бітів, починаючи з положення `p`, було би обернено на протилежний (тобто 1 замінено на 0 і навпаки), залишаючи решту бітів незмінними.

Вправа 2-8. Напишіть функцію `rightrot(x,n)`, яка повертає значення цілого `x`, оберненого в правий бік на `n` позицій.

2.10 Оператори та вирази присвоєння

Вираз на кшталт

```
i = i + 2
```

в якому змінна з лівого боку одразу повторюється на правому, можна записати в стислішій формі

```
i += 2
```

Оператор `+=` також називається оператором присвоєння. Більшість бінарних операторів (такі як `+`, що має лівий і правий операнд) мають відповідний оператор присвоєння *op* `=`, де *op* може бути одним з наступних

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

Якщо $expr_1$ і $expr_2$ — це два вирази, тоді

```
expr1 op= expr2
```

еквівалентно

```
expr1 = (expr1) op (expr2)
```

крім випадку, коли $expr_1$ обчислюється тільки один раз. Зверніть увагу на дужки навколо $expr_2$:

```
x *= y + 1
```

означає

```
x = x * (y + 1)
```

а не

```
x = x * y + 1
```

В якості прикладу наведемо функцію `bitcount`, яка обчислює кількість 1-бітів в своєму аргументі-цілому.

```
/* bitcount:  рахує 1-біти в x */
int bitcount(unsigned x)
{
    int b;

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}
```

Оголошення аргументу `x` як `unsigned` (беззнакового) гарантує, що коли його буде зміщено вправо, звільнені біти заповняться нулями, а не знаковими бітами, незалежно від машини на якій було запущено програму.

Крім стислості, оператори присвоєння мають переважають тим, що вони краще відповідають способу мислення людей. Ми кажемо «додати 2 до `i`» або «збільшити `i` на 2», а не «взяти `i`, додати 2, а потім помістити результат назад у `i`». Тому виразу `i += 2` надається перевага над `i = i+2`. На додаток, у складних виразах на зразок

```
yyval[yypv[p3+p4] + yypv[p1]] += 2
```

оператор присвоєння полегшує розуміння коду, оскільки читач не змушений ретельно перевіряти, що два довгих вирази дійсно відповідають один одному, або дивуватися, чому ні. Оператор присвоєння, може навіть допомогти компілятору виробити ефективніший код.

Ми вже бачили, що твердження присвоєння повертає певне значення і може вживатися у більших виразах; найтиповішим прикладом може бути

```
while ((c = getchar()) != EOF)
    ...
```

Інші оператори присвоєння (`+=`, `-=` тощо) також можуть зустрічатися всередині виразів, хоч і не так часто.

В усіх таких виразах, тип виразу присвоєння збігається з типом лівого операнду, а значення відповідає значенню після присвоєння.

Вправа 2-9. У двійковій числовій системі (two's complement number system), `x &= (x-1)` вилучає крайній правий біт з `x`. Поясніть чому. Скористайтеся з цього спостереження для написання швидшої версії `bitcount`.

2.11 Вирази умов

Вирази на зразок

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

обчислюють `z` як найбільше значення з-поміж `a` та `b`. *Умовний вираз* із використанням трійчастого оператора «`?:`» дають альтернативний спосіб написання подібної конструкції. У виразі

```
expr1 ? expr2 : expr3
```

вираз $expr_1$ розглядається першим. Якщо $expr_1$ ненульовий (істина), тоді обчислюється $expr_2$, і це й буде кінцевим значенням цілого виразу умови. У протилежному випадку, обчислюється $expr_3$, і він стане кінцевим значенням. Оцінюється тільки один із $expr_2$ та $expr_3$. Таким чином, щоб встановити `z` до максимального значення `a` або `b`, ми напишемо

```
z = (a > b) ? a : b;          /* z = max(a, b) */
```

Слід зауважити, що вираз умови — це дійсно вираз, і може бути використаний там де будь-який інший. Якщо $expr_2$ і $expr_3$ — різних типів, тип результату визначається правилами перетворення, обговорених раніше в цьому розділі. Наприклад, якщо `f` має тип `float`, а `n` — це `int`, тоді вираз

```
(n > 0) ? f : n
```

буде типу `float`, незалежно від того, чи `n` додатній.

Дужки не обов'язкові навколо першого виразу оскільки пріоритет `?` й `:` дуже низький, лишень трохи вищий за присвоєння. Проте, їх рекомендовано вживати, оскільки вони роблять умовну частину виразу очевиднішою.

Умовні вирази, як правило, призводять до стислішого коду. Наприклад, наступний цикл виводить `n` елементів масиву, по 10 на кожний рядок, стовпчики будучи розділені пробілом, кожний рядок (включаючи останній) завершується символом нового рядка.

```
for (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

Символ нового рядка виводиться після кожного десятого елемента і після n-нного. За рештою елементів слідує пробіл. Це може здаватися складним, але воно компактніше ніж еквівалент із if-else. Іншим хорошим прикладом є

```
printf("You have %d items%s.\n", n, n==1 ? "" : "s");
```

Вправа 2-10. Перепишіть функцію lower, яка обертає літери верхнього регістру на нижній, за використанням виразу умови замість if-else.

2.12 Пріоритет і послідовність обчислення

Таблиця 2.1 підбиває підсумок правил пріоритету й асоціативності всіх операторів, включаючи ті, що ми ще не обговорювали. Оператори, розміщені на одному рядку — рівні за пріоритетом; рядки знаходяться в порядку зменшення пріоритету, тож, наприклад, *, / та % мають однаковий пріоритет, який є вищим за, скажімо, бінарні + та -. «Оператор» () стосується виклику функції. Оператори -> із . використовуються для доступу до членів структур; їх буде розглянуто в Розділі 6 разом із функцією sizeof (яка визначає розмір об'єкту). В Розділі 5 обговорюється * (непряме звертання через покажчик) а також & (адреса об'єкту), тоді як в Розділі 3 розглянуто оператор-кому.

Зверніть увагу, що пріоритет розрядних операторів &, ^ та | нижчий за == та !=. З цього випливає, що вирази перевірки бітів на зразок

```
if ((x & MASK) == 0) ...
```

повинні обов'язково містити дужки, щоб добитися бажаного результату.

С, як і більшість мов, не уточнює послідовність в якій обчислюються операнди якогось оператора. (Виняток становлять &&, ||, ?: та ,.) Так, наприклад, у твердженні

```
x = f() + g();
```

f може бути обчислено до g або навпаки; тож, якщо функція f або g змінює якусь змінну, від якої залежить інша з цих функцій, тоді x також залежатиме від порядку обчислення. Проміжні результати можуть зберігатися в тимчасових змінних, щоб забезпечити певну послідовність. Аналогічно, порядок в якому розглядаються аргументи функцій, також не уточнюється, тож вираз

```
printf("%d %d\n", ++n, power(2, n));      /* ПОМИЛКА */
```

Табл. 2.1: Пріоритет і асоціативність операторів

Оператори	**Асоціативність**
`() [] -> .`	зліва направо
`! ~ ++ - + - * (`*тип*`) sizeof`	справа наліво
`* / %`	зліва направо
`+ -`	зліва направо
`<< >>`	зліва направо
`< <= > >=`	зліва направо
`== !=`	зліва направо
`&`	зліва направо
`^`	зліва направо
`|`	зліва направо
`&&`	зліва направо
`||`	зліва направо
`?:`	справа наліво
`= += -= *= /= %= &= ^= |= <<= >>=`	справа наліво
`,`	зліва направо

Унарні `&`, `+`, `-` та `*` мають вищий пріоритет за відповідні бінарні форми.

може видати відмінний результат у різних компіляторах, залежно від того, чи `n` збільшено до чи після виклику `power`. Вирішенням цього, звичайно, буде написати

```
++n;
printf("%d %d\n", n, power(2, n));
```

Виклики функцій, гніздовані присвоєння, а також оператори приросту та спаду зумовлюють «побічний ефект» — якась змінна міняє значення внаслідок обчислення виразу. В будь-якому виразі з побічним ефектом можуть бути тонкі відмінності щодо того в якій послідовності оновлюється значення змінних, що становлять частину виразу. Одну з нещасливих ситуацій можна зобразити наступним типовим прикладом:

```
a[i] = i++;
```

Питання полягає в тому, чи індекс вживається зі старим значенням, чи набув нового. Різні компілятори можуть тлумачити це по-різному, і видавати різні значення в залежності від інтерпретації. Стандарт навмисне не уточнює більшість таких питань. Випадки, коли всередині виразу має місце побічний ефект (присвоєння нового значення змінній), залишено на розсуд компілятора, оскільки найкраща послідовність дій залежить значною мірою від машинної архітектури. (Стандарт проте вказує, що всі побічні ефекти аргументів мають відбутися до того як викликано функцію, але це не допоможе при вищенаведеному виклику `printf`.)

Мораль зводиться до того, що написання коду, який залежить від порядку обчислення, вважається поганою програмістською практикою для будь-якої мови. Природньо, що треба знати речі, яких треба уникати, але, якщо ви не знатимете як щось працює на різних архітектурах, то і меншою буде ймовірність, що спокуситесь скористатися з переваг певної реалізації.

Розділ 3

Керування потоком

Вирази керування потоком якоїсь мови вказують на послідовність в якій виконуються обчислювальні дії. Ми вже ознайомилися з більшістю звичайних конструкцій керування потоком у попередніх прикладах; у цьому розділі ми поповнимо набір і будемо точнішими стосовно того, що обговорювалося раніше.

3.1 Вирази та блоки

Вираз на зразок `x = 0` або `i++` або `printf(...)` стають твердженням, якщо за ними слідує крапка з комою, як от

```
x = 0;
i++;
printf(...);
```

У C, крапка з комою позначає закінченням твердження, а не розділювач, як це прийнято в таких мовах як Pascal.

Фігурні дужки `{` та `}` використовуються для згуртовування оголошень і тверджень разом у *складне твердження*, або *блок*, тож вони синтаксично еквівалентні єдиному твердженню. Яскравим прикладом цього є фігурні дужки, що оточують твердження всередині функцій, так само як дужки після `if`, `else`, `while` або `for`. (Змінні може бути оголошено всередині будь-якого блоку; ми обговоримо це в Розділі 4.) Крапка з комою не потрібні після правої фігурної дужки, яка завершує блок.

3.2 If-else

Вирази `if-else` застосовуються для вираження ймовірних рішень. Формально синтаксис наступний:

```
if (вираз)
```

```
    твердження1
else
    твердження2
```

де `else`-частина не є обов'язковою. Вираз обчислюється; якщо він істинний (тобто вираз не має нульового значення), виконується *твердження*$_1$. Якщо хибний (вираз має нульове значення) і наявна `else`-частина, тоді виконується *твердження*$_2$, натомість.

Оскільки `if` просто перевіряє числове значення виразу, це дозволяє використовувати певні скорочення в кодуванні. Найочевиднішим є написання

```
if (вираз)
```

замість

```
if (вираз != 0)
```

Іноді це виглядає природньо і зрозуміло, а інколи занадто зашифровано.

Оскільки `else`-частина у виразі `if-else` не обов'язкова, то може виникнути неоднозначність, коли якийсь `else` пропущено у гніздованій послідовності виразів `if`. Це вирішується шляхом пов'язування `else` із найближчим попереднім `if`, що не має відповідного `else`. Так, наприклад, у

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

`else` відповідає `if`, що знаходиться всередині, що ми також підкреслили відступами. Якщо це не те, чого ви хотіли, тоді потрібно використати фігурні дужки, щоб змусити до відповідної належності:

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

Неоднозначність особливо небезпечна в ситуаціях як наступна:

```
if (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
else         /* ПОМИЛКА */
    printf("error -- n is negative\n");
```

Не зважаючи на те, що відступи ніби недвозначно вказують на те, що саме ви маєте на увазі, компілятор не збагне цього і пов'яже `else` з `if` всередині. Таку помилку буває дуже важко знайти пізніше, тому у випадку гніздованих `if` краще вживати фігурні дужки.

До речі, зверніть увагу на крапки з комою після `z = a`

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

Вони там тому, що граматично за `if` слідує твердження, а стверджувальні вирази на зразок `a = z;` завжди мають закінчуватися крапкою з комою.

3.3 Else if

Конструкція

```
if (вираз)
    твердження
else if (вираз)
    твердження
else if (вираз)
    твердження
else if (вираз)
    твердження
else
    твердження
```

зустрічаються настільки часто, що вимагають невеликого обговорення. Ця форма `if`-виразів є найбільш узагальненим способом написання виразів вибору з-поміж багатьох напрямків. Умови (вираз) розглядаються по-черзі, якщо якась з умов справджується,

твердження, пов'язані з цією умовою буде виконано, і це покладе кінець усьому ланцюжку. Звичайно, що код кожного виразу може складатися як з одного рядка, так і з групи, включеної у фігурні дужки.

Останнє `else` має справу з випадками, що не збіглися з жодною умовою вище, це те що виконуватиметься поза вибором, якщо всі умови не справдилися. Іноді може не бути жодної дії яку потрібно би було виконати поза вибором, у таких випадках кінцеве

```
else
    твердження
```

можна не включати, або його можна використати для вловлювання помилок на кшталт «неможливих» умов.

Для ілюстрації вибору у трьох напрямках, ось функція двійчастого пошуку, яка виявляє чи певне значення `x` з'являється у масиві `v`. Елементи `v` мають знаходитись у порядку зростання. Функція повертає позицію (число між 0 та `n-1`), якщо значення `x` знайдено у `v`, і -1 — якщо ні.

Двійчастий пошук спочатку порівнює (введене) значення `x` із середнім елементом масиву `v`. Якщо `x` менший за значення посередині, тоді пошук зосереджується на нижній частині списку, якщо навпаки — більший, то на верхній. В обох випадках, наступним кроком буде порівняння `x` із середнім елементом вибраної половини. Цей процес поділу відрізків на два продовжується доти, доки значення не буде знайдено, або діапазон виявиться порожнім.

```
/* binsearch: знаходить x серед v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low+high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else     /* знайдено збіг */
            return mid;
    }
    return -1;   /* збігу немає */
}
```

Фундаментальним рішенням є, чи x менший, більший, чи рівний середньому елементу v[mid] при кожному поступі; це дуже властиво else-if.

Вправа 3-1. Наш двійчастий пошук здійснює дві перевірки всередині циклу, тоді як однієї може вистачити (за умови додаткових перевірок поза циклом). Напишіть версію із одним тестуванням всередині циклу і порівняйте різницю в швидкості виконання програми.

3.4 Switch

Твердження switch здійснює вибір серед багатьох імовірностей шляхом перевірки, чи вираз збігається з одним із сталих цілочисельних значень, відповідно відгалужуючись.

```
switch (вираз) {
    case сталий-вираз: твердження
    case сталий-вираз: твердження
    default: твердження
}
```

Кожний випадок case помічено однією або більше цілочисельною сталою або сталим виразом. У випадку збігу зі значенням виразу, відбудеться виконання пов'язане з даною умовою. Усі вирази умов мають бути різними. Випадок, позначений як default, буде виконано, якщо жодна із попередніх умов не виявилася дійсною. default не є обов'язковим, якщо його не вказано і, якщо жоден з випадків не зійшовся — нічого не відбудеться. Випадки й уставна дія (default) можуть знаходитись у довільній послідовності.

У Розділі 1 ми написали програму для підрахунку кожної цифри, пробілів та інших знаків у тексті, використовуючи послідовність if ...else if ...else. Ось та сама програма зі switch:

```
#include <stdio.h>

main()   /* підраховує цифри, пробіли та інші знаки */
{
    int c, i, nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; i++)
        ndigit[i] = 0;
    while ((c = getchar()) != EOF) {
        switch (c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
```

```
            ndigit[c-'0']++;
            break;
        case ' ':
        case '\n':
        case '\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
        }
    }
    printf("digits =");
    for (i = 0; i < 10; i++)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n", nwhite, nother);
    return 0;
}
```

Вираз `break` спричиняє негайний вихід зі `switch`. Оскільки `case`-випадки служать тільки в якості ярликів, то після того, як код одного випадку виконано, виконання спадає вниз до наступного `case`, хіба що ви явно вкажете вийти. `break` і `return` — це найпоширеніші способи виходу зі `switch`. Вираз `break` так само може вживатися для того, щоб негайно покинути цикли `while`, `for` і `do` (ми обговоримо це пізніше у цьому розділі).

Низпадання крізь випадки `case` має хорошу і погану сторону. Добре це тим, що дозволяє декілька випадків прикріпити до однієї дії, як це сталося з числами у попередньому прикладі. Але це також означає, що за звичайних обставин, кожний окремий випадок case повинен закінчуватись break, щоб запобігти спаданню вниз до наступного. Спадання від одного випадку до іншого не є надійним, і схильне до розладу після зміни програми. За винятком складених ярликів для одного обчислення, низпадання слід використовувати помірно і коментувати.

Хорошим стилем вважається додання `break` після останнього випадку `case` (в попередньому прикладі це `default`), навіть якщо логічно в цьому немає потреби. Одного дня, коли буде додано інший `case` вкінці, ця невеличка хитрість може вас врятувати.

Вправа 3-2. Напишіть функцію `escape(s, t)`, яка перетворює такі знаки як новий рядок і крок табуляції на явні екрановані послідовності такі як `\n` та `\t` під час копіювання ланцюжка `t` до `s`. Застосуйте `switch`. Напишіть також зворотню функцію, яка би перетворювала екрановані послідовності знаків на дійсні знаки.

3.5 Цикли while та for

Ми вже зіткнулися з циклами `while` та `for`. У

```
while (вираз)
    твердження
```

вираз обчислюється. Якщо його значення не є нульовим, виконується *твердження* і *вираз* обчислюється знову. Цей цикл продовжуватиметься доти, доки *вираз* не набуде нульового значення, після чого виконання перейде у місце після *твердження*.

Твердження з `for`

```
for (вираз1; вираз2; вираз3)
    твердження
```

є рівнозначним

```
вираз1;
while (вираз2) {
    твердження
    вираз3;
}
```

за винятком поводження `continue`, яке описано у Розділі 3.7.

Граматично, всі три частини циклу `for` є виразами. Найчастіше $вираз_1$ та $вираз_3$ — це присвоювання або виклики функцій, тоді як $вираз_2$ — порівнювальний вираз. Будь-яку з трьох частин можна пропустити, але крапка з комою повинні залишитись. Якщо $вираз_1$ і $вираз_3$ опущено, тоді вони просто вилучаються з оцінювання. Якщо ж порівнювального $виразу_2$ бракує, уся конструкція набирає значення завжди істинної, тож

```
for (;;) {
    ...
}
```

є нескінченим циклом, який ймовірно буде перервано іншим способом, скажімо за допомогою `break` або `return`.

Використання `while` чи `for` значною мірою залежить від власних вподобань. Так, наприклад у

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ;  /* опустити пробіли */
```

немає ініціалізації або реініціалізації, тому `while` тут — природніший.

Виразам із `for` надається перевага коли присутні проста ініціалізація та приріст, оскільки `for` зберігає вирази, що контролюють цикл, ближче один до одного, у полі зору на вершечку циклу. Це найбільш очевидно у

```
for (i = 0; i < n; i++)
```

що являє собою ідіому C для обробки перших `n` елементів масиву, аналогічно `DO` у Fortran або `for` у Pascal. Ці аналогії, проте, не є досконалими, оскільки індекс та межа циклу for у C можуть змінюватися зсередини циклу, крім того індексна змінна `i` утримує своє значення після того, як цикл завершився з якоїсь причини. Оскільки компоненти `for` можуть складатися з довільних виразів, цикли `for` не обмежуються арифметичними прогресіями. Тим не менше, використання непов'язаних обчислень в ініціалізації та прирості `for` вважається поганим стилем, їх краще приберегти для керівної частини циклу.

В якості більшого прикладу, ось ще одна версія `atoi` — програми для конвертації рядка у свій числовий еквівалент. Цей приклад більш узагальнений ніж той, що ви знайдете у Розділі 2, він справляється з можливими пробілами попереду і можливими знаками `+` та `-`. (Розділ 4 містить `atof`, яка здійснює таке саме перетворення для чисел з рухомою точкою).

Структура програми відображає форму вводу:

```
пропустити пробіли, якщо такі є
здобути символ, якщо такий є
здобути десяткове значення і перетворити його
```

Кожний крок виконує тільки свою частину завдання і залишає решту незайманим для наступного кроку. Весь процес завершується на першому ж символі, що не може бути частиною числа.

```
#include <ctype.h>

/* atoi: перетворює s на ціле; 2-а версія */
int atoi(char s[])
{
    int i, n, sign;

    for (i = 0; isspace(s[i]); i++)   /* пропустити пробіли */
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')   /* пропустити знаки */
        i++;
    for (n = 0; isdigit(s[i]); i++)
```

```
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

Стандартна бібліотека передбачає складнішу функцію, `strtol`, для перетворення рядків у довгі цілі числа (загляніть до Розділу 5 Додатка Б).

Переваги зберігання керівних частин циклу разом ще очевидніші, коли маємо справу із декількома гніздованими циклами. Наступна функція — це сортування Шела, для впорядковування масиву цілих чисел. Головна ідея цього алгоритму сортування, який був винайдений Д.Л. Шелом у 1959 році, полягає в тому, що на початковій стадії порівнюються віддалені одне від одного елементи замість суміжних, на відміну від простіших взаємозамінних алгоритмів сортування. Це досить швидко усуває велику кількість невпорядкованості, тож пізнішим стадіям залишається менше роботи. Інтервал між порівнюваними елементами поступово зменшується до одного, в цьому пункті сортування перетворюється у звичайну методу переставляння прилеглих елементів.

```
/* shellsort: сортує v[0]...v[n-1] в послідовності зростання */
void shellsort(int v[], int n)
{
   int gap, i, j, temp;

   for (gap = n/2; gap > 0; gap /= 2)
      for (i = gap; i < n; i++)
         for (j = i - gap; j >= 0; && v[j] > v[j+gap]; j -= gap) {
             temp = v[j];
             v[j] = v[j+gap];
             v[j+gap] = temp;
      }
}
```

Ми бачимо три гніздованих цикли. Зовнішній контролює проміжок між порівнюваними елементами, звужуючи його починаючи з `n/2` у два рази при кожному поступі, доти, доки проміжок не стане нульовим. Середній цикл проходиться по кожному елементові. Внутрішній цикл порівнює кожну пару елементів розділених проміжком `gap` і переставляє ті, що не знаходяться в послідовності зростання. Оскільки проміжок `gap` поступово скорочується до одного, всі елементи зрештою мають остаточно впорядкуватись. Зауважте, як загальність конструкції `for` дозволяє зовнішньому циклу набрати тієї самої форми, що й інші, навіть якщо це не є арифметичною прогресією.

Ще один, останній, оператор С, це кома «,», яка найчастіше знаходить застосування у твердженнях `for`. Пара виразів, розділених комою, розцінюються у послідовності зліва направо. Тип і значення результату дорівнюватиме типу і значенню правого операнда. Таким чином, у твердженні `for` можна розмістити декілька виразів у різних

частинах, наприклад для обробки двох індексів одночасно. Це ілюстровано функцією `reverse(s)`, яка обертає ланцюжок `s` на місці.

```
#include <string.h>

/* reverce: обертає ланцюжок s на місці */
void reverse(char s[])
{
    int c, i, j;

    for (i = 0, j = strlen(s) - 1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Коми, що розділяють аргументи функцій чи змінні в оголошеннях тощо, не є операторами і не гарантують оцінювання зліва направо.

Кома-оператори повинні використовуватись вибірково. Вони найбільш придатні для близько споріднених конструкцій, як показано у циклі `for` функції `reverse`, також ви знайдете їх у макросах, там де багатоетапні обчислення треба занотувати одним виразом. Вираз з комою може бути доречним також для взаємозаміни елементів у `reverse`, там де заміна може розглядатися як одна операція:

```
for (i = 0, j = strlen(s) - 1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
```

Вправа 3-3. Напишіть функцію `expand(s1, s2)`, яка розкриває скорочення на кшталт `a-z` в ланцюжкові `s1` у відповідний повний список `abc...xyz` у `s2`. Дозвольте використання літер обох регістрів, а також цифр. Будьте готові до обробки таких випадків, як `a-b-c`, `a-z0-9` або `-a-z`. Добийтеся того, щоб «-» попереду або вкінці розглядалася буквально.

3.6 Цикли do-while

Як ми вже обговорили в Розділі 1, цикли `while` і `for` перевіряють умову завершення циклу на самому початку. На противагу першим двом, `do-while` перевіряє умову вкінці, після проходження кожного кроку у корпусі циклу; корпус завжди виконується щонайменше один раз.

Синтаксис `do` наступний:

```
do
    твердження
while (вираз)
```

Тож, спочатку буде виконано *твердження*, і лиш потім оцінено *вираз*. Якщо умова справджується, *твердження* виконується ще раз, і так далі... Коли вираз виявиться хибним, цикл буде перервано. За винятком змісту перевірки, `do-while` еквівалентний `repeat-untill` у Pascal.

Досвід свідчить, що `do-while` використовується значно рідше ніж `while` або `for`. Тим не менше, час од часу він - корисний, як у наступній функції `itoa`, яка перетворює число у символьний ланцюжок (функція, протилежна `atoi`). Ця задача трохи складніша, ніж може видатись з першого погляду, оскільки легкі способи генерації чисел розставляють ці числа в невірному порядку. Ми вирішили відтворити ланцюжок в оберненому порядку, а потім розвернути його.

```
/* itoa: перетворює n у знаки s */
void itoa(int n, char s[])
{
    int i, sign;

    if ((sign = n) < 0)         /* виявити знак */
        n = -n;                 /* перетворити n на додатнє */
    i = 0;
    do {       /* генерувати цифри в зворотньому порядку */
        s[i++] = n % 10 + '0';   /* одержати наступну цифру */
    } while ((n /= 10) > 0);     /* видалити її */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

Цей `do-while` — потрібний, або щонайменше зручний, оскільки принаймні один знак повинен потрапити в масив `s`, навіть якщо `n` буде нульовим. Ми використали фігурні дужки навколо єдиного твердження з якого складається корпус `do-while` (хоча вони не обов'язкові) для того, щоб поспішний читач не сприйняв `while`-частину за початок циклу `while`.

Вправа 3-4. У двійковому взаємодоповнюючому представленні чисел, наша версія `itoa` не справляється з найбільшим від'ємним числом, тобто значенням `n` що дорівнює $-(2^{-1})$. Поясніть чому ні. Поміняйте програму, щоб вона виводила це число правильно, незалежно від машини на якій вона виконується.

Вправа 3-5. Напишіть функцію `itob(n,s,b)` яка переводить десяткове `n` у `b` систему числення і зберігає це у рядку `s`. Так, наприклад, `itob(n,s,16)` сформує `n` як шістнадцяткове число, збережене в ланцюжку `s`.

Вправа 3-6. Напишіть версію `itoa`, яка братиме три аргументи замість двох. Третім аргументом буде мінімальна ширина поля. Конвертоване число буде доповнюватись пробілами зліва у разі потреби, щоб добитися певної ширини.

3.7 Break і continue

Часом зручно мати можливість вийти з циклу інакше, ніж шляхом тестування умови зверху або внизу. Саме твердження `break` забезпечує передчасний вихід із `for`, `while` або `do`-циклів, так само як і у випадку зі `switch`. Вираз `break` змушує найближчий цикл або `switch`, у якому він знаходиться, завершитись негайно.

Наступна функція, яку ми назвали `trim`, вилучає кінцеві пробіли, кроки табуляції та символи нового рядка в ланцюжкові, використовуючи `break` для виходу з циклу, як тільки знайдено перший знак, що не є ні пробілом, ні знаком табуляції, ні символом нового рядка.

```
/* trim: видаляє хвостові пробіли, табуляцію і символи нового рядка */
int trim(char s[])
{
    int n;

    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

`strlen` повертає довжину ланцюжка. Цикл `for` починає з кінця ланцюжка і перевіряє у зворотньому напрямку, шукаючи перший знак, який не є пробілом, табуляцією або символом нового рядка. Цикл перерветься, коли один з них буде знайдено або, коли `n` стане від'ємним (тобто, коли весь ланцюжок пройде перевірку). Вам слід перевірити, чи цей цикл веде себе правильно, навіть коли ланцюжок порожній або містить тільки символи пропуску.

Оператор `continue` — споріднений з `break`, але не так часто використовується. Він зумовлює початок наступної ітерації оточуючого циклу `for`, `while` або `do`. У випадку `while` і `do`, тестова частина виконується негайно; у випадку `for`, керування передається етапові приросту. Твердження `continue` вживається тільки з циклами, а не `switch`-конструкціями. Оператор `continue` всередині `switch`, розміщеному в циклі, спричинить наступну ітерацію циклу.

Як приклад, наступний фрагмент коду обробляє тільки додатні елементи масиву a, пропускаючи від'ємні.

```
for (i = 0; i < n; i++) {
    if (a[i] < 0)     /* пропустити від'ємні елементи */
        continue;
        ...           /* обробити додатні елементи */
}
```

Твердження з continue часто використовуються, коли наступна частина циклу занадто складна, тож перевірка протилежної умови та додатковий відступ коду вправо гніздить програму занадто глибоко.

3.8 Goto та мітки

C також включає один з операторів, яким часто зловживають — goto, а також мітки для переходу в дане місце виконання програми. Формально, goto ніколи не потрібний і практично майже завжди можна написати код, уникаючи його. Ми не використовували goto в цій книжці.

Проте, існує декілька ситуацій, коли goto може знайти собі застосування. Найпоширенішим є відмова від подальшого опрацьовування в якійсь глибоко гніздованій структурі, як наприклад вихід з двох або більше циклів одночасно. Оператора break не вистачить в таких випадках, оскільки він здійснює вихід тільки з найближчого циклу. Таким чином:

```
for ( ... )
    for ( ... ) {
        ...
        if (невдача)
            goto error;
    }
...

error:
    /* очистити безлад */
```

Така організація зручна, коли код обробки помилки не є простим, або коли помилки можуть статися в декількох місцях.

Позначки мають ту саму форму, що й назви змінних, але закінчуються двокрапкою. Вони можуть бути приєднані до будь-якого виразу, їхня функціональність схожа до goto. Областю дії мітки є ціла функція.

Як ще один приклад, розгляньмо завдання, що полягає у виявленні, чи два масиви a і b мають якийсь спільний елемент. Одним з можливих варіантів є:

```
for (i = 0; i < n; i++)
    for (j = 0; j < m; j++)
        if (a[i] == b[j])
            goto found;
   /* у протилежному випадку не знайдено */
   ...
found:
   /* знайдено: a[i] == b[j] */
   ...
```

Код з `goto` завжди можна написати без цього оператора, але напевне ціною додаткових тестів або додаткових змінних. Так, наприклад, пошук по масивах міг би виглядати як

```
found = 0;
for (i = 0; i < n && !found; i++)
    for (j = 0; j < m && !found; j++)
        if (a[i] == b[j])
            found = 1;
if (found)
    /* знайдено: a[i-1] == b[j-1] */
    ...
else
    /* не знайдено спільних елементів */
    ...
```

За декількома винятками, як от приклади наведені тут, код, який покладається на твердження `goto`, загалом, важче зрозуміти і підтримувати у робочому стані, ніж код без `goto`. Хоча ми не є догматиками в цьому питанні, все ж схоже, що оператори `goto` слід застосовувати дуже рідко, якщо й взагалі.

Розділ 4

Функції та структура програм

Функції розбивають великі обчислювальні завдання на менші, і дозволяють програмістам будувати на основі того, що написали інші, замість починати все з нуля. Хороші функції ховають деталі своєї роботи від частин програми, які не повинні про це знати, таким чином прояснюючи увесь код, і полегшуючи зміни.

С задумана, щоб зробити функції ефективними та легкими у використанні; програми на С, як правило, складаються з багатьох малих функцій, замість кількох великих. Сама програма може знаходитись в одному або декількох вихідних файлах. Вихідні тексти можна компілювати окремо і завантажити разом, поряд з попередньокомпільованими функціями бібліотеки. Ми не будемо заглиблюватись у цей процес тут, оскільки подробиці можуть відрізнятися в різних операційних системах.

Оголошення й означення функцій — це саме та область, де стандарт ANSI здійснив найбільші зміни в мові С. Як ми вже бачили у Розділі 1, тепер з'явилася можливість опису типу аргументів під час оголошення функцій. Синтаксис визначення функції також змінився, тож оголошення та визначення збігаються. Це дозволяє тепер компілятору вловити набагато більше помилок, ніж раніше. Більше того, коли аргументи оголошено належним чином, відповідні поправки типів здійснюються автоматично.

Стандарт прояснює правила області дії імен, зокрема він вимагає, щоб існувало тільки одне визначення для кожного зовнішнього об'єкту. Ініціалізація стала більш загальною: тепер можна започатковувати автоматичні масиви і структури.

Препроцесор С також вдосконалено. Нові засоби препроцесора тепер включають повніший набір умовних компіляційних директив, спосіб створення залапкованих ланцюжків з аргументів макросу, і покращене керування над процесом розкриття макросів.

4.1 Основні знання про функції

Для початку, спланууймо та напишемо програму виводу кожного рядка вводу, який містить певний «зразок» або ланцюжок знаків. (Своєрідну імітацію програми `grep` Unix.) Так, наприклад, пошук зразка, що складається з літер «`ould`» у наборі рядків

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

видав би нам

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

Цю роботу можна акуратно розбити на три частини:

```
while (ще є рядок)
    if (рядок містить зразок)
        вивести рядок
```

Безперечно, ми могли би помістити увесь код такої програми в `main`, але кращим виходом буде використати цю структуру на свою користь, розбивши кожну її частину на окремі функції. З малими частинами набагато легше впоратись, ніж з однією великою, оскільки деталі, що не стосуються справи, можна сховати всередині функцій, і можливість небажаних взаємодій буде зведено нанівець. Також, частини можуть виявитись корисними для інших програм.

«Ще є рядок» — це `getline` — функція, яку ми написали у Розділі 1 і «вивести рядок» — це `printf`, яку хтось створив вже для нас. Це означає, що нам залишилось написати функцію, яка вирішувала б, чи рядок містить ланцюжок, який збігається зі зразком.

Ми можемо вирішити це завдання шляхом написання функції `strindex(s,t)`, яка повертає положення, або індекс у ланцюжку `s`, де починається ланцюжок `t`, або -1, якщо `s` не містить `t`. Оскільки масиви в C починаються з позиції 0, їхні індекси можуть бути або нульовими, або додатними, тож від'ємне число, як от -1 — зручне для сигналізування невдачі. Якщо нам пізніше потрібне буде більш витончене порівняння зі зразком, ми можемо замінити тільки функцію `strindex`, тоді як решта коду може залишитись незмінною. (Стандартна бібліотека передбачає функцію `strstr`, аналогічну `strindex`, за винятком того, що вона повертає покажчик замість індексу.)

Маючи усе спланованим, заповнення деталей програми — досить прямолінійне. Нижче наведено увесь код, тож ви можете побачити, як частини взаємодіють разом. Поки що, зразок, за яким здійснюється пошук, складається з буквенного ланцюжка, що не є найуніверсальнішим механізмом. Згодом, ми дійдемо до обговорювання того, як ініціювати символьні масиви, а в Розділі 5 ми продемонструємо, як зробити зі зразка параметр, який можна задати під час обігу програми. Ця версія `getline` трохи відрізняється від використаної в Розділі 1; можливо буде повчальним, якщо ви порівняєте їх.

```
#include <stdio.h>
#define MAXLINE  1000    /* максимальна довжина рядків */

int getline(char line[], int max);
int strindex(char source[], char searchfor[]);

char pattern[] = "ould";  /* зразок, який шукатиметься */

/* знаходить всі рядки, що збігаються зі зразком */
main()
{
    char line[MAXLINE];
    int found = 0;

    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            printf("%s", line);
            found++;
        }
    return found;
}

/* getline: розміщає рядок у s, повертає довжину */
int getline(char s[], int lim)
{
    int c, i;

    i = 0;
    while (--lim > 0 && (c=getchar() != EOF && c != '\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}

/* strindex: вертає індекс t у s, або -1, якщо t не знайдено */
int strindex(char s[], char t[])
{
    int i, j, k;

    for (i = 0; s[i] != '\0'; i++) {
        for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
```

```
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

Кожне визначення функції має форму

```
тип_повернення назва_функції(оголошення аргументів)
{
    оголошення та твердження
}
```

Окремих частин може не бути. Мінімальною функцією є

```
dummy() {}
```

яка не виконує жодної дії і не повертає жодного значення. Такі бездіяльні функції часом корисні в якості заповнювача під час розробки програми. Якщо тип повернення опущено, припускається `int`.

Програма — це лишень набір визначень змінних і функцій. Комунікація між функціями відбувається через аргументи та значення, повернуті функціями, а також через зовнішні змінні. Функції можуть знаходитись у будь-якій послідовності у вихідному тексті, а сам вихідний код програми можна розділити на численні файли за умови, що жодна функція не розділена.

Твердження `return` — це механізм повернення значення викликаною функцією тому, хто її викликав. За `return` може слідувати будь-який вираз:

```
return вираз;
```

Вираз буде перетворено до типу повернення функції, якщо потрібно. Навколо *виразу* часто вживаються дужки, але вони не обов'язкові.

Викликова функція має право ігнорувати значення, повернене викликаною. Більше того, вираз після `return` взагалі може бути відсутній, у разі чого викликовій функцій жодного значення повернено не буде. Керування над виконанням програми також повертається викликовій без жодного значення, коли виконання «спадає з кінця» функції, досягши крайньої фігурної дужки. Це не заборонено, але ймовірно є проявом несправності, якщо функція повертає значення в одному місці, і жодного — в іншому. В будь-якому разі, якщо функції не вдалося повернути значення, то її «значення» напевне складається з непотрібу.

Наша програма пошуку зразка повертає з `main` статус — число знайдених збігів. Це значення стає доступним для використання середовищем, яке викликало програму.

Механізм компіляції і завантаження програми на C, розбитої на численні файли, відрізняється в різних системах. У системі UNIX, наприклад, це завдання може виконати команда `cc`, згадана у Розділі 1. Припустімо, що три функції збережено у трьох різних файлах під назвою `main.c`, `getline.c` і `strindex.c`. У такому разі команда

```
cc main.c getline.c strindex.c
```

скомпілює ці три файли, розміщуючи об'єктний код, отриманий в результаті, у `main.o`, `getline.o` і `strindex.o`, після чого завантажує їх всіх до виконавчого файла під назвою `a.out`. Якщо виникла якась помилка, скажімо у `main.c`, цей файл можна перекомпілювати окремо, а результат завантажити з попередніми об'єктними файлами командою

```
cc main.c getline.o strindex.o
```

Команда `cc` користується умовними назвами «`.c`» на противагу «`.o`» для того, щоб розрізнити вихідний файл від об'єктного.

Вправа 4-1. Напишіть функцію `strindex(s,t)`, яка би повертала крайнє праве положення ланцюжка `t` у `s`, або -1, якщо `t` не знайдено.

4.2 Функції, які не повертають цілих

Досі, наші приклади функцій або не повертали жодного значення (`void`), або повертали `int`. А що, як функція має повернути якийсь інший тип? Багато математичних функцій, таких як `sqrt`, `sin` або `cos` повертають `double`; інші спеціалізовані функції повертають інші типи. Щоб проілюструвати, як впоратись з цією задачею, напишімо та випробуємо функцію `atof(s)`, яка перетворює ланцюжок `s` у її еквівалент у вигляді числа з рухомою точкою подвійної точності. Функція `atof` — це розширення `atoi`, версії якої ми розглянули у Розділі 2 і 3. Остання обробляла можливий знак і десяткову крапку, так само як брак чи наявність цілої або дробової частини. Наша версія не є високоякісною функцією перетворення вводу; це зайняло би більше місця, ніж ми хотіли би використати тут. Стандартна бібліотека включає `atof`; у заголовку `<stdlib.h>` ви знайдете її оголошення.

Перш за все, сама `atof` має оголосити тип значення, яке вона повертає, оскільки це — не `int`. Назва типу стоятиме перед назвою функції:

```
#include <ctype.h>

/* atof: перетворює s у число подвійної точності */
double atof(char s[])
```

```
{
    double val, power;
    int i, sign;

    for (i = 0; isspace(s[i]); i++) /* пропустити пробіли */
        ;
sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')
        i++;
    for (val = 0.0; isdigit(s[i]); i++)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        i++;
    for (power = 1.0; isdigit(s[i]); i++) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10.0;
    }
    return sign * val / power;
}
```

По-друге, так само важливо, щоб викликова функція знала, що `atof` повертає не-`int` значення. Один із способів досягти цього — це явно оголосити `atof` у викликовій функції. Оголошення, показане у наступній примітивній програмці-калькуляторі (достатньої хіба що для збалансовування чекової книжки), яка читає по одному числу на рядок, з можливим знаком попереду, і додає їх, виводячи поточну суму після кожного вводу:

```
#include <stdio.h>

#define MAXLINE 100

/* простенький калькулятор */
main()
{
    double sum, atof(char []);
    char line[MAXLINE];
    int getline(char line[], int max);

    sum = 0;
    while (getline(line, MAXLINE) > 0)
        printf("\t%g\n", sum += atof(line));
    return 0;
}
```

Оголошення

```
double sum, atof(char []);
```

вказує на те, що `sum` є змінною типу `double` і, що `atof` — це функція, яка візьме один аргумент `char[]` і поверне число типу `double`.

Функцію `atof` має бути оголошено та визначено без протиріч. Якщо сама `atof` та її виклик всередині `main` матимуть неузгоджені типи у тому самому вихідному тексті, компілятор помітить помилку. Але (що буває частіше), якщо `atof` було скомпільовано окремо, розбіжність виявлено не буде, `atof` повернула би `double`, яке `main` розглянула би як `int`, видаючи беззмістовне значення.

Зважаючи на сказане нами, що оголошення мають збігатися з визначеннями, це може здатися дивним. Причиною невідповідності може бути відсутність прототипу, функцію непрямо оголошено під час першої її появи у виразі, як от

```
sum += atof(line);
```

Якщо назва, якої не було попередньо оголошено, з'явиться у якомусь виразі, і за нею слідує ліва дужка, її буде розглянуто в контексті як оголошення назви функції, а функції, як відомо, без задання повертають `int` без жодних припущень щодо її аргументів. Більше того, якщо оголошення функції не містить аргументів, як наприклад

```
double atof();
```

це також означає, що нічого не допускається стосовно аргументів `atof`, і перевірку параметрів буде вимкнено. Цей особливий зміст порожнього списку аргументів передбачено для того, щоб дозволити старшим С-програмам компілюватися з новими компіляторами. Але використовувати порожній список у нових програмах — це погана ідея. Якщо функція бере аргументи, оголосіть їх, якщо ні — скористайтеся з `void`.

Маючи правильно оголошену `atof`, ми могли би написати `atoi` (функцію перетворення ланцюжка на `int`), виходячи з неї:

```
/* atoi: за допомогою atof перетворює ланцюжок s на ціле число */
int atoi(char s[])
{
    double atof(char s[]);

    return (int) atof(s);
}
```

Зверніть увагу на структуру оголошень і на твердження `return`. Значення виразу

```
return вираз;
```

перетворюється до типу функції, до того, як відбудеться повернення. Таким чином, значення `atof`, число подвійної точності, автоматично перетворено на `int` при появі в `return`, оскільки функція `atoi` має повернути `int`. Ця операція, проте, потенційно, відкидає частину інформації, тож деякі компілятори можуть попередити про це. Оператор зведення типів заявляє відверто, що саме це малося на увазі, і пригнічує будь-які попередження.

Вправа 4-2. Продовжте `atof`, щоб вона могла оперувати експоненційним представленням у формі

```
123.45e-6
```

де за числом з рухомою точкою може слідувати `e` або `E` і необов'язкова знакова експонента.

4.3 Зовнішні змінні

Програма на C складається також з набору зовнішніх об'єктів — змінних або функцій. Прикметник «зовнішній» використовується як протиставлення «внутрішньому», який описує аргументи та змінні, визначені всередині функцій. Зовнішні змінні — означені поза межами будь-якої функції і, таким чином, потенційно доступні багатьом функціям. Самі функції завжди залишаються зовнішніми, оскільки C не дозволяє означати функції всередині інших функцій. Стандартно, зовнішні змінні і функції мають одну властивість, що полягає у тому, що всі звертання до них за тим самим ім'ям — навіть з функцій, скомпільованих окремо — посилаються на ту саму річ. (Стандарт називає це *зовнішньою сполучністю*.)

У цьому сенсі, зовнішні змінні схожі до блоків `COMMON` мови Fortran або змінних з найвіддаленішого блока в Pascal. Ми дізнаємось пізніше, як означувати зовнішні змінні та функції, видимі тільки з того самого вихідного файла.

Оскільки зовнішні змінні доступні глобально, вони служать альтернативою аргументам функцій, і можуть використовуватись для обміну даними між функціями. Будь-яка функція може звернутися до зовнішньої змінної, посилаючись на її ім'я, якщо це ім'я якимось чином оголошено.

Якщо функції змушені поділяти велике число змінних між ними, зовнішні змінні зручніші й ефективніші, ніж довгі списки аргументів. Але, як вже було вказано у Розділі 1, ці міркування треба застосовувати з обережністю, оскільки вони можуть погано сказатися на структурі програми, і призвести до програм із завеликою кількістю сполучень даних між функціями.

Зовнішні змінні також корисні через більшу область дії і тривалість життя. Автоматичні змінні є внутрішніми для функцій; вони з'являються при вході у функцію і зникають при виході з неї. На противагу, зовнішні змінні — постійні, а отже утримують значення від одного виклику функції до іншого. Таким чином, якщо дві функції мусять спільно використовувати якісь дані, але жодна з них не викликає іншої, часто найзручніше, щоб спільні дані зберігалися у зовнішніх змінних, замість передачі їх туди-сюди у вигляді аргументів.

Дослідімо це питання глибше на більшому прикладі. Завдання полягатиме в написанні програми-калькулятора, яка б забезпечувала операторами `+`, `-`, `*` і `/`. Оскільки це легше втілити - калькулятор використовуватиме зворотній польський запис замість інфіксного (звичайного). (Зворотня польська нотація використовується у деяких кишенькових калькуляторах а також у таких мовах як Forth або Postscript.)

У зворотньому польському записі, кожний оператор слідує за операндами; інфіксний вираз

```
(1 - 2) * (4 + 5)
```

буде введено як

```
1 2 - 4 5 + *
```

Дужки зайві; нотація буде недвозначною доти, доки ми знаємо скільки операндів кожний оператор очікує.

Втілення — просте. Кожний операнд проштовхується у стек; коли надходить оператор, належне число операндів (двоє — для бінарних операторів) виштовхуються зі стеку, і застосовано оператор; результат проштовхується назад у стек. У прикладі вище, скажімо, 1 і 2 проштовхуються, а потім замінюються наслідком їхнього віднімання: -1. Згодом, 4 і 5 проштовхуються, і замінюються 9. Результат множення -1 на 9, що дорівнюватиме -9, замінить і їх у стеку. Нарешті, коли досягнуто кінця рядка вводу, значення з верхівки стеку виштовхується і виводиться на екран.

Таким чином, структура програми складатиметься із циклу, що виконуватиме відповідну дію, залежно від оператора чи операнда, що надійшов:

```
while (наступний оператор чи операнд не є кінцем файла)
    if (число)
        проштовхнути його
    else if (оператор)
        виштовхнути операнди
        виконати дію
        проштовхнути результат
    else if (новий рядок)
        виштовхнути і вивести верхівку стеку
    else
```

```
помилка
```

Операції проштовхування і виштовхування зі стеку прості, але, як тільки додається перевірка помилок і відновлення, код стає занадто довгим, тож краще помістити кожну з них в окрему функцію, щоб можна було повторити той самий код протягом виконання програми. Також потрібна окрема функція для добування наступного введеного операнда чи оператора.

Основним питанням розробки, до якого ми ще не дійшли, є місцезнаходження стеку, тобто — які функції мають прямий доступ до нього? Одне з можливих рішень — зберегти його в `main`, і передавати стек і поточне положення в ньому функціям, які проштовхуватимуть і виштовхуватимуть його дані. Але `main` не повинна знати про змінні, які керують стеком, вона має здійснювати тільки операції проштовхування і виштовхування. Тож ми вирішили зберегти стек і відповідну інформацію у зовнішніх змінних, доступних функціям проштовхування і виштовхування, а не в `main`.

Перекласти цю схему у код досить легко. Якщо, ми вважатимемо поки що, що наша програма перебуває у одному вихідному файлі, вона матиме приблизно наступний вигляд:

```
#include (файли, які включаються)
#define  (означення)

оголошення функцій для main

main() { ... }

зовнішні змінні для проштовхування (push) і виштовхування (pop)

void push( double f ) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }
функції, викликані getop
```

Пізніше ми розглянемо, як це можна розбити на два або більше вихідних файли.

Функція `main` складатиметься з циклу, що міститиме великий перемикач (`switch`) типу оператора чи операнда; це — типовіше використання `switch` ніж те, що ми бачили у Розділі 3.4.

```
#include <stdio.h>
#include <stdlib.h>    /* для atof() */

#define MAXOP   100    /* максимальний розмір операнда або оператора */
```

```
#define NUMBER  '0'    /* сигналізувати, що номер знайдено */

int getop(char []);
void push(double);
double pop(void);

/* калькулятор зі зворотньою польською нотацією */
main()
{
    int type;
    double op2;
    char s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            if (op2 != 0.0)
                push(pop() / op2);
            else
                printf("error: zero divisor\n");
            break;
        case '\n':
            printf("\t%.8g\n", pop());
            break;
        default:
            printf("error: unknown command %s\n", s);
            break;
        }
    }
```

```
    return 0;
}
```

Оскільки + й * це переставні (комутативні) оператори, послідовність у якій виштовхнуті операнди поєднуються не має значення, але у випадку - та /, правий і лівий операнд повинні розрізнятися. Наприклад, у

```
push(pop() - pop());          /* НЕПРАВИЛЬНО */
```

послідовність, в якій два виклики pop обчислено, не визначено. Щоб забезпечити правильну послідовність, потрібно виштовхнути перше значення у тимчасову змінну, як ми це зробили в main:

```
#define MAXVAL    100    /* максимальна глибина стеку */

int sp = 0;              /* наступна вільна позиція у стеку */
double val[MAXVAL];      /* стек значень */

/* push: проштовхування у стек значень */
void push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        printf("error: stack full, can't push %g\n", f);
}

/* pop: виштовхнути і повернути верхнє значення зі стеку */
double pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        printf("error: stack empty\n");
        return 0.0;
    }
}
```

Змінна вважається зовнішньою, якщо її означено поза межами будь-якої функції. Таким чином, стек та індекс стеку, які спільно використовуватимуться push і pop, визначено зовні цих функцій. Але сама main не звертається до стеку або положення у стеку, тож представлення може бути прихованим.

Тепер, звернімо нашу увагу на втілення getop — функції, що видобуває наступний оператор чи операнд. Це просте завдання. Пропустити пробіли і табуляцію. Якщо наступний символ не скидається на число або десяткову точку, повернути його. У протилежному випадку, зібрати ланцюжок чисел (які можуть включати десяткову точку) і повернути NUMBER, яке сигналізуватиме, що число — успішно здобуто.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getop: отримати наступний знак або числовий операнд */
int getop(char s[])
{
    int i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    s[1] = '\0';
    if (!isdigit(c) && c != '.')
        return c;           /* не є числом */
        i = 0;
    if (isdigit(c))         /* зберегти частину, що є цілим */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')           /* зберегти дробову частину */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}
```

Що таке getch і ungetch? Часто бувають випадки, коли програма не може визначити, що вона прочитала досить вводу доти, доки не прочитано забагато. Один з прикладів — коли програма збирає знаки, що складають число, і поки не зустрінеться перший нецифровий знак, число не вважатиметься повним. Але тоді програмою прочитано один зайвий знак, до якого вона не була готова.

Таку проблему можна було б вирішити, якби була можливість скасування прочитаного небажаного знака. Тоді, кожний раз, як програма прочитає на один символ більше, вона зможе відкинути його назад на ввід, тож решта коду поводитиметься так, ніби цього знака ніколи прочитано не було. На щастя, скасування здобуття знака

досить легко імітувати шляхом написання пари кооперуючих функцій. Тож, getch подає наступний знак вводу на розгляд, тоді як ungetch повертає його назад перед тим, як прочитати новий ввід.

Як вони співпрацюють разом — не складно. Функція ungetch поміщає виштовхнутий знак у спільно використовуваний буфер — символьний масив. Функція getch читає з буфера, якщо там щось є, і викликає getchar, якщо буфер порожній. Має існувати також індексна змінна, яка би реєструвала положення поточного знака в буфері.

Оскільки, для getch і ungetch, буфер та індекс — спільні, і повинні зберігати свої значення між викликами, то вони мають були зовнішніми для обох функцій. Ми можемо написати getch і ungetch із зовнішніми змінними, як:

```
#define BUFSIZE 100

char buf[BUFSIZE];      /* буфер для ungetch */
int bufp = 0;           /* наступна вільна позиція у buf */

int getch(void)         /* отримати (можливо виштовхнутий) знак */
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

void ungetch(int c)     /* виштовхнути знак назад у ввід */
{
    if (bufp >= BUFSIZE)
        printf("ungetch: too many characters\n");
    else
        buf[bufp++] = c;
}
```

Стандартна бібліотека включає функцію ungetch, яка дає можливість виштовхнути один знак; ми розглянемо її у Розділі 7. Ми ж використали масив для виштовхнутих знаків, замість одного єдиного, щоб продемонструвати загальніший підхід.

Вправа 4-3. Маючи основний каркас, розширте програму-калькулятор. Додайте оператор частки (%) і можливість використання від'ємних чисел.

Вправа 4-4. Додайте команди для виводу верхніх елементів стеку без їхнього виштовхування, копіювання цих елементів і перестановки місцями перших двох. Додайте команду очистки стеку.

Вправа 4-5. Додайте доступ до таких функцій бібліотеки, як sin, exp і pow. Подивіться <math.h> у Додатку Б, Розділі 4.

Вправа 4-6. Додайте команди для оперування змінними. (Це не складно, надати двадцять шість змінних з назвами, що складаються з однієї літери.) Додайте змінну для найостаннішого виведеного значення.

Вправа 4-7. Напишіть функцію `ungets(s)`, яка би виштовхувала цілий ланцюжок у ввід. Чи повинна `ungets` знати про `buf` і `bufp`, чи їй достатньо лише використовувати `ungetch`?

Вправа 4-8. Припустімо, що ніколи не буде більш ніж одного виштовхнутого знака. Змініть `getch` і `ungetch` відповідно.

Вправа 4-9. Наші `getch` і `ungetch` не справляються належним чином із виштовхнутим `EOF`. Вирішіть, яким буде їхнє поводження у випадку виштовхнутого `EOF`, після чого перепишіть їх.

Вправа 4-10. Альтернативною організацією програми буде використання `getline` для зчитування цілого ввідного рядка; це робить `getch` і `ungetch` зайвими. Переробіть калькулятор із застосуванням цього підходу.

4.4 Правила області дії

Функції і зовнішні змінні, що складають програму на C, не обов'язково повинні бути скомпільованими одночасно; вихідний текст програми може зберігатися у декількох файлах і попередньо-компільовані функції можуть бути завантаженими з бібліотек. Серед питань, які нас можуть зацікавити, є

- Як написати оголошення, тож змінні оголошено належним чином під час компіляції?
- Як оголошення організовано, тож усі частини буде з'єднано як слід під час завантаження програми?
- Як оголошення організовано, тож існуватиме лише одна копія?
- Як започаткувати зовнішні змінні?

Давайте обговоримо ці питання шляхом реорганізації нашої програми-калькулятора у декілька файлів. З практичної точки зору, калькулятор — це занадто маленька програма, щоб розбивати її, але вона служить гарним прикладом питань, що виникають у більших програмах.

Областю дії назви є та частина програми, в якій цю назву можна використовувати. У випадку автоматичних змінних, оголошених на початку функції, областю дії буде функція, в якій цю назву було оголошено. Локальні змінні з тією самою назвою, але у різних функціях не мають жодних стосунків одна з одною. Те саме чинне і для параметрів функцій, які насправді також є локальними змінними.

Область дії зовнішньої змінної або функції триває, починаючи з моменту їх було оголошено і аж до кінця скомпільованого файла. Наприклад, якщо `main`, `sp`, `val`, `push` і `pop` означено у одному файлі, у послідовності, вказаній нижче:

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

```
int sp = 0;
double val[MAXVAL];

void push(double f) { ... }

double pop(void) { ... }
```

тоді змінні `sp` і `val` можуть бути використаними у `push` і `pop` просто шляхом виклику їхньої назви; додаткові оголошення зайві. Але ці назви не бачить `main`, так само вона не бачить `pop` і `push`.

З іншого боку, якщо звернутися до зовнішньої змінної до того, як її означено або, якщо її означено у відмінному вихідному файлі від того, де вона використовується, тоді є обов'язковим оператор `extern`.

Важливо розрізняти оголошення зовнішніх змінних і їхні визначення. Оголошення заявляє про властивості змінної (головним чином її тип); визначення також виділяє простір для зберігання. Якщо рядки

```
int sp;
double val[MAXVAL];
```

з'являться зовні будь-якої функції, вони визначають зовнішні змінну `sp` і `val`, виділять місце для їхнього зберігання, також служать оголошенням для решти даного вихідного файла. З іншого боку, рядки

```
extern int sp;
extern double val[];
```

оголошують для решти вихідного файла, що `sp` становить `int` і `val` є масивом типу `double` (чий розмір визначено десь у іншому місці), але вони не створюють самих змінних чи виділяють місце для їхнього зберігання. Кожний файл, що входить в програму, повинен містити тільки одне визначення зовнішніх змінних; інші файли можуть містити оголошення з `extern`, щоб дістатися до них. (Можливі також оголошення `extern` у самому файлі, що містить ці визначені змінні.) Розміри масивів повинні бути вказаними з визначеннями, але не обов'язкові з `extern`-оголошеннями.

Ініціалізація зовнішньої змінної відбувається тільки за умови, що існує її визначення.

Хоча це не характерно для такої програми, але функції `push` і `pop` могли би бути визначені у одному файлі, тоді як змінні `val` і `sp` визначено і ініційовано у іншому. В такому разі, такі визначення і оголошення потрібно би було зв'язати разом:

у першому файлі:

```
extern int sp;
extern double val[];

void push(double f) { ... }

double pop(void) { ... }
```

у другому файлі:

```
int sp = 0;
double val[MAXVAL];
```

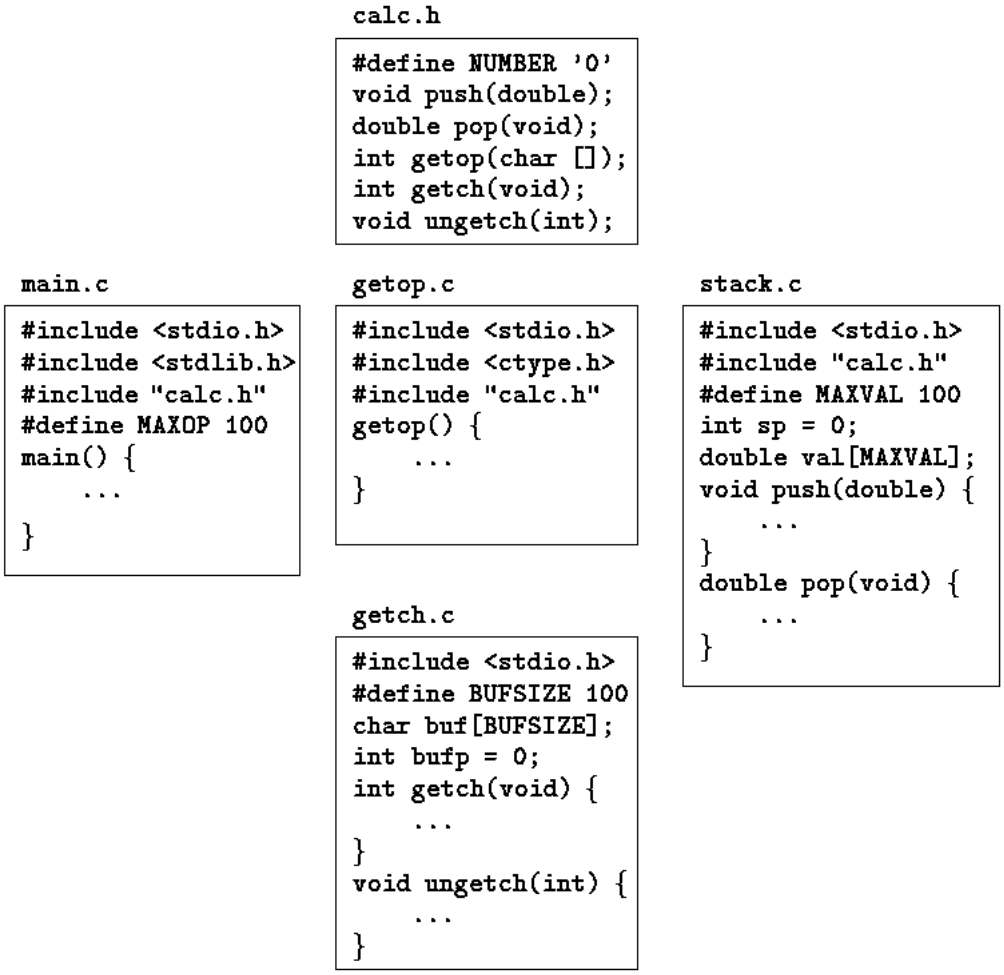
Оскільки оголошення `extern` у першому файлі знаходяться попереду і зовні визначень функцій, вони стосуються всіх функцій. Одного оголошення вистачить для цілого першого файла. Така сама організація була би потрібною, якби `sp` і `val` слідували за своїм використанням у тому самому файлі.

4.5 Файли заголовка

Давайте розглянемо тепер поділ програми-калькулятора на декілька вихідних файли, так ніби кожна з її складових була значно більшою. Функція `main` опинилася би в одному файлі, який ми назвали би `main.c`; `push`, `pop` і їхні змінні розміщено би у другому, `stack.c`; `getop` у третьому — `getop.c`. Ми розділили їх одне від одного, оскільки вони складали би окремо скомпільовану бібліотеку у справжній програмі.

Існує ще одна річ, про яку варто потурбуватися — це визначення і оголошення, спільно використовувані цими файлами. Наскільки це можливо, ми би хотіли зосередити ці речі у одному місці, тож існуватиме тільки одна копія, яку буде прочитано і яку слід підтримувати у робочому стані під час розвитку програми. Відповідно, ми помістимо цей спільний матеріал у файлі заголовка `calc.h`, який буде включено у разі потреби. (Рядок `#include` описано у Розділі 4.11.) Отримана в результаті програма може виглядати так:

Треба знайти компроміс між бажанням, щоб кожний файл мав доступ тільки до інформації, потрібної для здійснення свого завдання і практично дійсністю того, що важко підтримувати у робочому стані багато файлів заголовка. У випадку програм помірного розміру, напевне, найкраще мати один файл заголовка, який міститиме спільні дані для інших частин програми; саме це рішення ми продемонстрували тут. Для більших програм, може виникнути потреба в суттєвішій організації і більшій кількості файлів заголовка.



4.6 Статичні змінні

Змінні `sp` і `val` зі `stack.c` а також `buff` і `bufp` із `getch.c` існують для приватного використання у функціях відповідних вихідних файлів і не призначені для доступу чимось іншим. Оголошення `static` зовнішньої змінної або функції обмежує зону дії цього об'єкту рештою вихідного файла, який буде скомпільовано. Зовнішній `static`, таким чином, дає можливість приховати такі назви як `buf` і `bufp` у комбінації з `getch-ungetch`, які в свою чергу, теж повинні бути зовнішніми для спільного їхнього використання, але чиї подробиці не повинні бути видимими користувачам `getch` і `ungetch`. Статичний тип зберігання даних вказується шляхом додання слова `static` перед звичайним оголошенням. Якщо дві функції і дві змінні скомпільовано у одному файлі,

як от

```
static char buf[BUFSIZE];    /* буфер для ungetch */
static int bufp = 0;         /* наступне вільне положення в buf */

int getch(void) { ... }

void ungetch(int c) { ... }
```

тоді жодна інша функція не в змозі буде звернутися до `buf` і `bufp` і ці назви не створюватимуть конфлікту з такими самими назвами в інших файлах тієї самої програми. Аналогічно, змінні, які використовуються функціями `push` і `pop` для маніпулювання стеком, можна приховати, якщо оголосити `sp` і `val` як `static`.

Зовнішні статичні змінні найчастіше вживаються зі змінними, але їх можна застосовувати також до функцій. Звично, назви функцій є глобальними, тобто їх видно з будь-якої частини програми. Проте, якщо функцію оголошено як `static`, її назва стане невидимою поза межами файла, в якому її було оголошено.

Означення `static` застосовується також щодо внутрішніх змінних. Внутрішні статичні змінні будуть локальними для даної функції, так само як автоматичні змінні, але на відміну від останніх, вони залишатимуться дійсними, зберігаючи своє значення, замість появи і зникнення кожного разу як функцію викликано. Це означає, що внутрішні статичні змінні надають приватне, постійне збереження даних у межах однієї функції.

Вправа 4-11. Змініть `getop`, таким чином, щоб їй не потрібно було використовувати `ungetch`. Підказка: застосуйте внутрішню статичну змінну.

4.7 Регістрові змінні

Оголошення регістрової змінної вказує компілятору, що дана змінна буде інтенсивно використовуватись. Ідея полягає у тому, що регістрові змінні поміщено у регістри процесору, що призводить до менших і швидших програм. Але компілятори вільні ігнорувати цю пораду. Оголошення регістрових змінних може виглядати як

```
register int x;
register char c;
```

і так далі. Регістрові оголошення можуть застосовуватись до автоматичних змінних і до формальних параметрів функції. У останньому випадку, це може мати вигляд

```
f(register unsigned m, register long n)
{
    register int i;
```

```
    ...
}
```

На практиці, на регістрові змінні накладено обмеження, які відображають можливості підставового устаткування. Тільки декілька змінних з кожної функції можуть бути збереженими в регістрах і тільки певні типи дозволяються. Надмір регістрових оголошень не є шкідливим, тим не менш, оскільки слово `register` ігноровано для зайвих або заборонених оголошень. Адресу регістрової змінної неможливо отримати (отримання адрес змінних розглянуто у Розділі 5), незалежно від того, чи змінну справді розміщено у регістрі, чи ні. Обмеження кількості і типів регістрових змінних відрізняється на різних машинах.

4.8 Структура блоків

C не є блоково-структурованою мовою на зразок Pascal чи інших мов, оскільки функцію не можна означити всередині іншої функції. З іншого боку, змінні можуть бути описані у блок-структурованому вигляді всередині функції. Оголошення змінних (включаючи ініціалізацію) може слідувати за лівою фігурною дужкою, яка започатковує будь-яке складене твердження, а не тільки починає функцію. Змінні, оголошені таким чином, приховано від однаково названих змінних у зовнішніх блоках і залишаються існувати до відповідної правої фігурної дужки. Наприклад, у

```
if (n > 0) {
    int i;          /* оголошення нової i */

    for (i = 0; i < n; i++)
        ...
}
```

зоною дії змінної `i` є «істинне» відгалуження `if`; це не матиме жодного стосунку до жодного і поза межами цього блока. Автоматичну змінну, оголошену та ініційовану у блоці, ініційовано кожний раз при входженні у цей блок.

Автоматичні змінні, включаючи формальні параметри функцій, також приховані від зовнішніх змінних і функцій з тією самою назвою. Маючи оголошення

```
int x;
int y;

f(double x)
{
    double y;
}
```

змінна x, як параметр функції f типу double, немає нічого спільного з зовнішньою x типу int. Те саме стосується змінної y. Але, загалом, краще уникати назв змінних, що збігаються з назвами з інших зон дії, занадто велика ймовірність плутанини і помилок.

4.9 Ініціалізація

Про ініціалізацію мимоходом вже згадувалося багато разів, але завжди периферійно, як частина іншої теми. Цей розділ підводить підсумок деяких правил, після того як ми оговорили різноманітні типи зберігання. У випадку відсутності явної ініціалізації, зовнішні і статичні змінні буде гарантовано надано значення нуль. Автоматичні і регістрові змінні матимуть невизначене початкове значення (тобто непотріб).

Скалярні змінні може бути ініційовано під час їхнього визначення шляхом додачі до їхньої назви знака рівності і виразу:

```
int x = 1;
char squota = '/';
long day = 1000L * 60L * 60L * 24L;  /* мілісекунд у день */
```

Для зовнішніх і статичних змінних, ініціалізатор повинен бути сталим виразом; ініціалізація відбувається один раз, концептуально до того як програма починає виконуватись. Для автоматичних і регістрових змінних, ініціалізатор не обов'язково повинен бути константою: це може бути будь-який вираз, включаючи попередньо-визначенні значення, навіть виклики функцій. Так, наприклад, ініціалізацію в програмі бінарного пошуку з Розділу 3.3 може бути написано як

```
int binsearch(int x, int v[], int n)
{
    int low = 0;
    int high = n - 1;
    int mid;
    ...
}
```

замість

```
int low, high, mid;

low = 0;
high = n - 1;
```

Ефективно, ініціалізація автоматичних змінних, це просто скорочення для виразів присвоєння значення. Якій формі надавати перевагу, це значною мірою є питанням смаку. Ми загалом використовували явні присвоєння, оскільки ініціалізатори у оголошеннях важче побачити і знаходяться далі від місця використання.

Масив може бути ініційовано через додання до оголошення списку ініціалізаторів, включених у фігурні дужки і розділених комою. Як приклад — ініціалізація масиву `days` з кількістю днів кожного місяця:

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

Коли розмір масиву не вказано, компілятор обчислить його, перелічивши ініціалізатори, 12 у цьому випадку.

Якщо надано менше ініціалізаторів, ніж вказаний розмір масиву, решті буде присвоєне значення нуль у випадку зовнішніх, статичних і автоматичних змінних. Помилка виникне у випадку зайвих ініціалізаторів. Не існує способу вказати повторні ініціалізатори, так само як елемент посередині масиву, без вказівки попередніх значень також.

Символьні масиви — це спеціальний випадок ініціалізаторів: можна використати ланцюжок замість нотації з фігурних дужок і ком:

```
char pattern[] = "ould";
```

що є скороченням для довшого, але рівнозначного

```
char pattern[] = { 'o', 'u', 'l', 'd', '\0' };
```

В обох випадках, масив складатиметься з п'яти елементів: чотири літери і кінцевий `'\0'`.

4.10 Рекурсія

Функції у C можуть використовуватись рекурсивно; тобто, функція може викликати сама себе безпосередньо або опосередковано. Розглянемо вивід числа як символьний ланцюжок. Як ми вказали раніше, цифри генеруються у неправильній послідовності: молодші числа доступні раніше за старші числа, хоча їх потрібно виводити навпаки.

Існує два способи розв'язання цієї проблеми. Одним буде збереження цифр у масив під час їхнього генерування, потім видрукувати їх у зворотній послідовності, як ми це здійснили з `itoa` у Розділі 3.6. Альтернативою є рекурсивне вирішення, у якому `printd` спочатку викликає себе щоби впоратись з початковими цифрами, після чого виводить кінцеві цифри. Майте на увазі, що ця версія може зазнати невдачі на найбільшому від'ємному числі.

```
#include <stdio.h>

/* printd:  вивести n як десяткове */
void printd(int n)
{
    if (n < 0) {
        putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
        putchar(n % 10 + '0');
}
```

Під час виклику функцією самої себе рекурсивно, кожне звернення отримує свіжий набір усіх автоматичних змінних, незалежних від попереднього набору. Таким чином, у `printd(123)`, перша `printd` отримує аргумент `n = 123`. Вона передає `12` другому `printd`, яка в свою чергу передає `1` третьому. `printd` третього рівня виводить `1` після чого повертається до `printd` другого рівня, яка виводить `2`, потім повертається до першого рівня. Він виводить `3` і завершується.

Інший хороший приклад рекурсії, це `quicksort`, алгоритм сортування, розроблений C.A.R. Hoare у 1962 році. Маючи масив, вибирається один елемент і решта розбивається на дві групи — ті, що менші за вибраний елемент і ті, що більше або дорівнюють йому. Той самий процес потім застосовано рекурсивно до цих двох груп. Коли підгрупа залишилась з менше ніж двома елементами, вона не потребує жодного сортування; це зупиняє рекурсію.

Наша версія `quicksort` не є найшвидшою з можливих, але це одна з найпростіших. Ми використовуємо середній елемент кожної частини масиву для подальшого поділу.

```
/* qsort:    сортує v[left]...v[right] у послідовності зростання */
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right)          /* нічого не робити, якщо масив   */
        return;                 /* містить менше ніж два елементи */
    swap(v, left, (left + right)/2); /* перемістити елемент поділу */
    last = left;                                      /* до v[0] */
    for (i = left + 1; i <= right; i++)              /* поділити */
        if (v[i] < v[left])
            swap(v, ++last, i);
```

```
    swap(v, left, last);                /* відновити елемент поділу */
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Ми перемістили операцію переставляння в окрему функцію `swap`, оскільки вона відбувається три рази в `qsort`.

```
/* swap:     поміняти місцями v[i] і v[j] */
void swap(int v[], int i, int j)
{
    int temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Стандартна бібліотека включає власну версію `qsort`, спроможну сортувати об'єкти будь-якого типу. Рекурсія може займати місце, оскільки десь потрібно втримувати стек значень для обробки. Також вона не є швидшою. Але рекурсивний код компактніший і часто набагато простіший у написанні і розумінні, чим нерекурсивний еквівалент. Рекурсія особливо зручна для рекурсивно-визначених даних на зразок деревовидних структур; ми побачимо гарний приклад цього у Розділі 6.6.

Вправа 4-12. Пристосуйте ідеї `printd` для написання рекурсивної версії `itoa`; тобто перетворіть ціле у ланцюжок, шляхом виклику рекурсивної функції.

Вправа 4-13. Напишіть рекурсивну версію функції `reverse(s)`, яка обертає ланцюжок `s` на місці.

4.11 Препроцесор C

C забезпечує певними додатковими мовними засобами через посередництво препроцесора, який концептуально є окремим першим кроком в компіляції. Дві, риси, якими найчастіше користуються, це `#include` — для включення вмісту іншого файла під час компіляції і `#define` — для заміни лексеми певною послідовністю знаків. Іншими властивостями, розглянутими у цьому розділі, є обумовлена компіляція і макроси з аргументами.

4.11.1 Включення файлів

Включення файлів полегшує використання наборів `#define` і оголошень (серед інших речей). Будь-який рядок вихідного тексту, котрий має форму

```
#include "filename"
```

або

```
#include <filename>
```

замінюється на вміст файла *filename*. Якщо назву файла взято у лапки, його пошук, типово, почнеться з того місця, де перебуває вихідний текст програми; якщо його там не знайдено, або якщо назву файла включено у < й >, пошук слідує правилам, визначеним системою, щоб знайти його. Включений файл може й самий містити рядки `#include`.

Часто, ви знайдете декілька рядків `#include` на початку вихідного файла, для включення загальних тверджень `#define` і оголошень `extern`, або для доступу до оголошень прототипів функцій для бібліотечних функцій з таких заголовків як <`stdio.h`>, наприклад. (Якщо бути точним, заголовки не обов'язково повинні бути файлами; деталі щодо звернення до заголовків визначаються системою).

`#include` — це переважний спосіб зв'язування разом оголошень у великих програмах. Він гарантує, що всі вихідні файли забезпечено тими самими визначеннями і оголошенням змінних, усуваючи можливість особливо неприємних вад. Звичайно, якщо включений файл змінено, то всі файли, які залежать від нього, слід перекомпілювати.

4.11.2 Заміна макросів

Визначення макросу має форму

```
#define назва текст-заміни
```

Це вимагає найпростішої заміни макросу — наступні появи лексеми *«назва»* буде замінено *«текстом заміни»*. Назва в `#define` має ту саму форму, що й назви змінних; текст заміни може бути довільним. Зазвичай текст заміни розміщено на тому самому рядкові, але якщо треба продовжити його на наступних, додайте зворотню похилу \ укінці кожного рядка, який ви хочете продовжити. Зона дії назви, означеної `#define`, починається з пункту, де її було означено і аж до кінця файла, який буде скомпільовано. Визначення можуть вживати попередні визначення. Заміни матимуть місце тільки для лексем і не відбуваються всередині ланцюжків взятих у лапки. Скажімо, якщо `YES` — це означена назва, заміна стане неможливою у випадку `printf("YES")` або `YESMAN`. Назву можна пов'язати з будь-яким текстом заміни. Наприклад

```
#define    forever        for(;;)   /* нескінчений цикл */
```

визначає для нескінченого циклу нове слово — `forever`.

Можна також означити макроси з аргументами, тож текст заміни зможе відрізнятися для різних викликів макросу. Для прикладу, створимо макрос під назвою `max`:

```
#define    max(A, B) ((A) > (B) ? (A) : (B))
```

Хоч це й виглядає ніби виклик функції, виклик `max` розкривається у вбудований код. Кожна поява формального параметра (у цьому випадку `A` й `B`) замінюватиметься на відповідні дійсні аргументи. Таким чином, рядок

```
x = max(p+q, r+s);
```

замінюється на

```
x = ((p+q) > (r+s) ? (p+q) : (r+s));
```

Доки аргументи вживаються несуперечливо, цей макрос служитиме для будь-якого типу даних; немає потреби у відмінному `max` для іншого типу даних, як це відбувається у функціях.

Якщо ви розглянете розкриття `max`, то можете помітити деякі пастки. Вирази обчислено двічі; це може бути погано, якщо вони включатимуть побічні ефекти, такі як оператори приросту або ввід і вивід. Так, наприклад,

```
max(i++, j++);      /* НЕПРАВИЛЬНО */
```

здійснює приріст більшого значення двічі. Треба також звертати увагу на дужки, щоб упевнитись, що порядок обчислень збережено; уявіть, що станеться якщо макрос

```
#define square(x)          x * x    /* НЕПРАВИЛЬНО */
```

викликати як `square(z+1)`.

Не зважаючи на це, макроси корисні. Один з прикладів їхнього використання на практиці можна знайти в <`stdio.h`>, де `getchar` і `putchar` часто означено як макроси, щоб запобігти втраті робочого часу на виклик функції для кожного опрацьованого знака. Функції у <`ctype.h`> також часто втілено як макроси.

Назви можна скасувати за допомогою `#undef`, зазвичай, щоб упевнитись, що якась функція є справді функцією а не макросом:

```
#undef getchar

int getchar(void) { ... }
```

Формальні параметри не замінюються, якщо їх було взято у лапки при означенні макросу. Проте, якщо попереду назви параметра стоїть знак # у тексті заміни, комбінація розшириться до взятого в лапки ланцюжка, у якому параметр заміниться на фактичний аргумент. Це можна комбінувати зі зчепленням ланцюжків, наприклад для відлагодження макросу виводу:

```
#define    dprint(expr)        printf(#expr " = %g\n", expr)
```

Якщо його викликати як

```
dprint(x/y)
```

макрос розкриється у

```
printf("x/y" " = %g\n", x/y);
```

ланцюжки буде зчеплено, тож отримаємо

```
printf("x/y = %g\n", x/y);
```

Всередині отриманого аргументу, кожний знак ¨ замінюється на \¨ і кожний \ на \\, тож результат буде допустимою ланцюжковою константою.

Оператор препроцесора ## дає можливість зчепити дійсні аргументи під час розкриття макросу. Якщо параметр у тексті заміни є суміжним з ##, цей параметр буде замінено на дійсний аргумент, ## і оточуючі пробіли усунуто і результат переглянуто. Наприклад, наступний макрос `paste` зчеплює власні два аргументи:

```
#define    paste(front, back)  front ## back
```

тож `paste(name, 1)` створить лексему `name1`.

Правила гніздованого використання ## досить заплутані; подальші деталі ви знайдете у Додатку A.

Вправа 4-14. Означте макрос `swap(t,x,y)`, який міняє місцями два аргументи типу `t`. (Тут поможе блокова структура коду.)

4.11.3 Обумовлене включення файлів

Існує також можливість керувати самою первинною обробкою — за допомогою умовних виразів, які обробить препроцесор. Це дає можливість вибіркового включення коду, залежно від значень умов, розглянутих під час компіляції.

Рядок `#if` обчислює константний цілочисельний вираз (який не може включати `sizeof`, зведення типів або констант енумерації). Якщо вираз не є нульовим, всі наступні рядки до `#endif` або `#elif` або ж `#else` буде включено. (Препроцесорний вираз `#elif` подібний до `esle-if`.) Вираз `defined(`*`назва`*`)` після `#if` дорівнюватиме 1, якщо назву було означено і 0, якщо ні.

Наприклад, щоб упевнитись, що вміст файла `hdr.h` включено тільки один раз, сам вміст оточується умовою на зразок наступної:

```
#if !defined(HDR)
#define HDR

/* тут йде вміст hdr.h */

#endif
```

Перше включення `hdr.h` визначає ім'я `HDR`; наступні включення знаходитимуть це ім'я означеним і перестрибуватимуть далі аж до `#endif`. Цей стиль можна використати і щоб запобігти багаторазового включення тих самих файлів. Якщо послідовно впроваджувати даний стиль, тоді кожний файл заголовка може самостійно включати будь-які інші файли заголовка, від яких він залежить, звільняючи користувача заголовка від клопоту про взаємозалежності.

Наступні декілька рядків перевіряють змінну `SYSTEM` для того, щоб вирішити, яку саме версію файла заголовка включити:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
    #include HDR
```

Вирази `#ifdef` і `#ifndef` є спеціалізованою формою, яка перевіряє чи ім'я означено, чи ні. Перший приклад з `#if` вище можна було також написати як

```
#ifndef HDR
#define HDR

/* тут йде вміст hdr.h */
```

```
#endif
```

Розділ 5

Покажчики та масиви

Покажчиком є змінна, яка містить адресу іншої змінної. Покажчики часто використовуються в C, частково тому, що це іноді єдиний спосіб виразити обчислення, та частково тому, що вони, як правило, призводять до компактнішого і ефективнішого коду, ніж той, що можна написати іншим способом. Покажчики та масиви тісно пов'язані між собою; цей розділ також досліджує їхній взаємозв'язок і демонструє як скористатися з нього.

Покажчики іноді поєднують з твердженнями `goto`, як чудовий спосіб створення програм, які неможливо зрозуміти. Це правда, якщо користуватися ними недбало. Так само досить легко створити покажчики які вказують кудись непередбачувано. Однак, за певної дисципліни, завдяки покажчикам можна досягти ясності та простоти. Саме цю їхню рису ми намагатимемось проілюструвати.

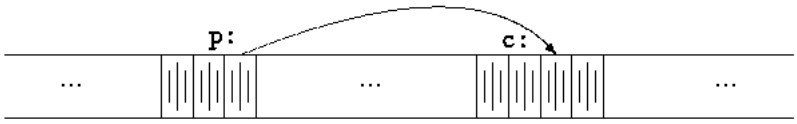
Основною зміною ANSI C було внесення ясності щодо того, як можна маніпулювати покажчиками, фактично роблячи обов'язковим те, що хороші програмісти практикують, і до чого хороші компілятори примушують. На додаток, тип `void *` (порожній покажчик) замінює `char *`, як належний тип загального покажчика.

5.1 Покажчики й адреси

Почнімо зі спрощеного зображення того як організовано пам'ять. Типова машина має масив послідовно нумерованих або адресованих комірок (секцій) пам'яті, якими можна орудувати окремо або прилеглими групами. Поширеним випадком є, коли один байт може складати `char`, тоді як пара однобайтових комірок розглядається як коротке ціле (`short int`), а чотири суміжних байти утворюють довге ціле. Покажчик — це група комірок (часто дві або чотири), що можуть утримати адресу. Тож, якщо `c` — це `char`, а `p` — це покажчик, що вказує на адресу `c`, то ми можемо графічно зобразити цю ситуацію як наступне:

Унарний оператор `&` добуває адресу об'єкта, тож твердження

```
p = &c;
```



присвоює адресу `c` змінній `p`, і `p`, як говорять, «вказує на» `c`. Оператор `&` застосовується лише з об'єктами з пам'яті — змінними й елементами масивів. Його неможливо використати з виразами, константами або регістровими змінними.

Унарний `*` є оператором непрямого звертання або «розіменування». Коли його застосовано до покажчика, він дає доступ до об'єкта, на який вказує покажчик. Припустімо, що `x` та `y` є цілими, а `ip` — це покажчик на `int`. Наступний штучний приклад демонструє як оголосити покажчик, і як користуватися `&` та `*`:

```
int x = 1, y = 2, z[10];
int *ip;             /* ip є покажчиком на int */

ip = &x;             /* ip тепер вказує на x */
y = *ip;             /* y дорівнює тепер 1 */
*ip = 0;             /* x дорівнює тепер 0 */
ip = &z[0];          /* ip тепер вказує на z[0] */
```

Оголошення змінних `x`, `y` та `z` нам зрозуміле. Оголошення ж покажчика `ip`

```
int *ip;
```

задумане як мнемоніка (як символічне); воно вказує на те, що вираз `*ip` є типу `int`. Синтаксис оголошення змінної імітує синтаксис виразів, в яких змінна може з'явитися. Ця сама система застосовується також при оголошенні функцій. Наприклад,

```
double *dp, atof(char *);
```

вказує на те, що вирази `*dp` і `atof(s)` повертають значення типу `double` і, що аргументом `atof` є покажчик на `char`.

Ви також повинні звернути увагу на те, що покажчик обмежений вказувати тільки на окремий рід об'єктів: кожний покажчик вказує на певний тип даних. (Існує один виняток — «покажчик на `void`» (порожній покажчик), використовуваний для утримування будь-якого типу покажчиків, але до якого неможливо непрямо звернутися. Ми повернемося до нього у Розділі 5.11.)

Якщо `ip` вказує на ціле `x`, тоді `*ip` може з'являтися в будь-якому контексті, в якому може `x`, тож

```
*ip = *ip + 10;
```

збільшує `*ip` на 10.

Унарні оператори `*` та `&` прив'язані тісніше ніж арифметичні оператори, тож присвоєння

```
y = *ip + 1;
```

візьме те, на що вказує `ip`, додасть 1 і присвоїть отриманий результат `y`, тоді як

```
*ip += 1;
```

здійснює приріст того, на що вказує `ip`; те саме стосується

```
++*ip;
```

та

```
(*ip)++;
```

В останньому прикладі, дужки обов'язкові; без них, вираз збільшив би саму `ip` (тобто адресу) замість того, на що вона вказує, оскільки унарні оператори такі як `*` й `++` асоціюються (спрягаються) справа наліво.

І, нарешті, оскільки покажчики, це також змінні, їх можна виживати без непрямого звертання. Наприклад, якщо `iq` — це інший покажчик на `int`, то

```
iq = ip;
```

копіює вміст `ip` до `iq`, таким чином примушуючи `iq` вказувати на той самий об'єкт, на який вказує `ip`.

5.2 Покажчики й аргументи функцій

Оскільки C передає тільки значення аргументам функцій, не існує прямого способу для викликаної функції змінити значення змінної викликової. Для прикладу, певна функція сортування може обмінюватися двома невпорядкованими аргументами з функцією під назвою `swap`. Недостатньо написати

```
swap(a, b);
```

там де функцію `swap` визначено як

```
void swap(int x, int y)    /* НЕПРАВИЛЬНО */
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

Отримуючи тільки значення, `swap` не може вплинути на самі змінні a `i b` у функції, яка викликала `swap`. Рутина, наведена вище, міняє місцями тільки копії `a` та `b`.

Щоб досягти бажаного ефекту, потрібно, щоб викликова програма передала покажчики на значення, які потрібно поміняти:

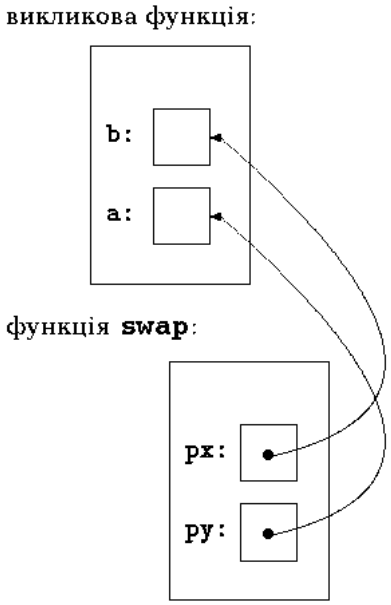
```
swap(&a, &b);
```

Оскільки оператор `&` добуває адресу змінної, `&a` буде покажчиком на `a`. А в самій функції `swap`, параметри необхідно оголосити як покажчики, і через їхнє посередництво дістатися самих операндів.

```
void swap(int *px, int *py)  /* міняє місцями *px і *py */
{
    int temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

Графічно:



Аргументи-покажчики забезпечують функцію доступом до об'єктів викликової функції, і можливістю їх змінювати. Як приклад, уявіть собі функцію `getint`, яка здійснює перетворення вводу вільного формату, розбиваючи потік знаків на цілі величини, по одному цілому числу на один виклик функції. `getint` повинна повернути обчислене значення а також сигналізувати кінець файла, коли ввід закінчився. Ці значення потрібно передати назад різними шляхами, оскільки незалежно від того, яке значення використовується для `EOF`, це також могло би бути чинним значенням введеного цілого.

Одне з рішень, це щоб `getint` повертала вказівник кінця файла як кінцеве значення самої функції, одночасно використовуючи аргументи-покажчики для збереження перетворених цілих у викликовій функції. Саме ця схема використовується також у `scanf` (дивіться Розділ 7.4).

Наступний цикл заповнює масив цілими викликаючи `getint`:

```
int n, array[SIZE], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) != EOF; n++)
    ;
```

Кожний виклик присвоює `array`[`n`] значення наступного цілого, знайденого у вводі, і нарощує `n`. Зверніть увагу, що суттєво вказати саме адресу `array`[`n`] функції `getint`, інакше `getint` не має способу передати перетворене ціле назад викликовій функції.

Наша версія `getint` повертає `EOF` у випадку кінця файла, нуль — якщо наступний знак вводу не є цілим, і додатнє значення — якщо ввід містить чинне ціле число.

```
#include <ctype.h>
```

```
int getch(void);
void ungetch(int);

/* getint: зберігає наступне введене ціле у *pn */
int getint(int *pn)
{
    int c, sign;

    while (isspace(c = getch()))    /* опустити пробіли */
        ;
    if (!isdigit(c) && c != EOF && c != '+' && c != '-') {
        ungetch(c);     /* c не є числом */
        return 0;
    }
    sign = (c == '-') ? -1 : 1;
    if (c == '+' || c == '-')
        c = getch();
    for (*pn = 0; isdigit(c), c = getch())
        *pn = 10 * *pn + (c - '0');
    *pn *= sign;
    if (c != EOF)
        ungetch(c);
    return c;
}
```

Скрізь у `getint`, `*pn` використовується як звичайна змінна типу `int`. Ми також використали `getch` і `ungetch` (описані у Розділі 4.3), тож один зайвий знак, який ми змушені були прочитати, можна було відкинути назад на ввід.

Вправа 5-1. Так як ми написали, функція `getint` сприймає `+` або `-` без числа як чинне числове значення. Виправте цю помилку, щоб таку послідовність було відкинуто.

Вправа 5-2. Напишіть `getfloat`, аналогічну `getint` функцію але для чисел з рухомою точкою. Який тип повертає `getfloat`?

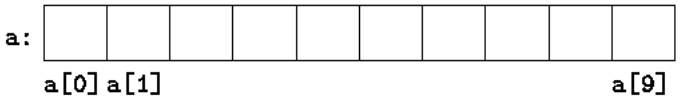
5.3 Покажчики та масиви

У C існує тісний зв'язок між покажчиками і масивами, настільки тісний, що покажчики і масиви повинні обговорюватись одночасно. Будь-яку дію, що можна здійснити через індексацію масиву, можна також через покажчики. Версія з покажчиками буде дещо швидшою, але, принаймні для непосвячених, також трохи важчою у розумінні.

Оголошення

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

означує масив розміром у 10 елементів, тобто блок з 10-и суміжних об'єктів з назвами a[0], a[1], . . . , a[9].



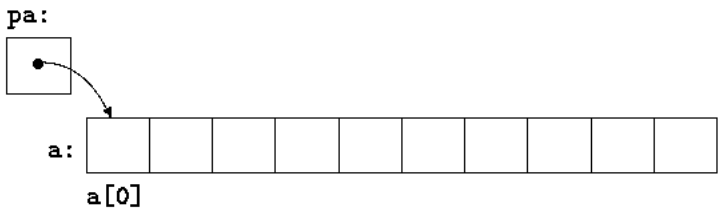
Позначення a[i] означає i-ний елемент масиву. Якщо pa, це покажчик на ціле, оголошений як

```
int *pa;
```

тоді присвоєння

```
pa = &a[0];
```

змушує pa вказувати на елемент з індексом нуль масиву a, тобто pa міститиме адресу a[0].



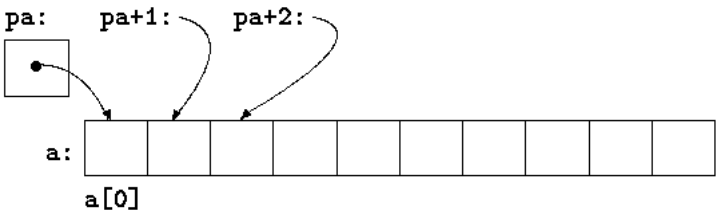
Тепер присвоєння

```
x = *pa;
```

копіює вміст a[0] до x.

Якщо pa вказує на певний елемент масиву, тоді певна річ, що pa+1 вказуватиме на наступний елемент, тоді як pa+n вказуватиме на елемент n після pa, тоді як pa-n — на елемент n попереду. Таким чином, якщо pa вказує на a[0], тоді

```
*(pa+1)
```



посилається на вміст a[1], pa+n є адресою a[n], а *(pa+n) — вмістом a[n].

Ці зауваження дійсні, незалежно від типу або розміру змінних масиву a. Зміст виразу «додати 1 до покажчика», і взагалі вся арифметика покажчиків зводиться до того, що pa+1 вказує на наступний об'єкт, тоді як pa+n вказує на n-ний об'єкт поза pa.

Відповідність між індексацією й арифметикою покажчиків дуже близька. За визначенням, значення змінної або виразу типу «масив» — це адреса елемента нуль масиву. Таким чином, після присвоєння

```
pa = &a[0];
```

pa й a матимуть абсолютно однакові значення. Оскільки назва масиву — це лише синонім місцезнаходження початкового елемента, присвоєння pa=&a[0] можна з таким самим успіхом написати як

```
pa = a;
```

Швидше несподіваним, з першого погляду, може здатися той факт, що посилання на a[n] можна також записати як *(a+n). Обчислюючи a[n], C сама перетворює його в *(a+n) — ці дві форми еквівалентні. Застосовуючи оператор & до обох частин цієї еквівалентності, ми дійдемо висновку, що &a[n] й a+n також однакові: a+n є адресою n-ного елемента після a. З іншого боку, якщо pa є покажчиком, він міг би використовуватись з індексом: pa[n], що рівнозначно *(pa+n). Одним словом, вираз масив-та-індекс рівнозначний тому, що записано як покажчик і зміщення.

Існує одна відмінність між назвою масиву і покажчиком, яку слід пам'ятати. Справа в тім, що покажчик — це змінна, тож pa=a і pa++ дозволено. Проте назва масиву не є мінною, тож конструкції на зразок a=pa або a++ заборонено.

Коли функції передано назву масиву, що передається насправді, це місцезнаходження першого елемента. Всередині викликаної функції цей аргумент стає локальною змінною, тож назва масиву, як параметр, насправді є покажчиком — тобто змінною, котра містить адресу. Ми можемо використати цей факт для написання іншої версії strlen — функції, яка обчислює довжину ланцюжка.

```
/* strlen:  повертає довжину ланцюжка s */
int strlen(char *s)
```

```
{
    int n;

    for (n = 0; *s != '\0', s++)
        n++;
    return n;
}
```

Оскільки `s` — це покажчик, то приріст його дозволено; `s++` не матиме жодного впливу на символьний ланцюжок функції, яка викликала `strlen` — лише здійснює приріст приватної копії покажчика в `strlen`. Це означає, що всі виклики на зразок

```
strlen("hello, world");         /* ланцюжкова стала */
strlen(array);                  /* char array[100]; */
strlen(ptr);                    /* char *ptr; */
```

працюватимуть.

Формально, при визначенні функції, параметри

```
char s[];
```

і

```
char *s;
```

рівнозначні. Ми надаємо перевагу останньому, оскільки він відвертіше вказує на те, що ця змінна є покажчиком. Коли назву масиву передано функції, остання може, залежно від того як їй зручніше, вважати, що їй передано або масив, або покажчик, і опрацьовувати його відповідно. Вона навіть може вживати обидві нотації, якщо це здається їй слушним і зрозумілим.

Можливо також передати функції лише частину масиву, вказавши покажчиком на початок частини масиву. Наприклад, якщо `a` — це масив, то

```
f(&a[2])
```

і

```
f(a+2)
```

обидва, передають функції `f` адресу частини масиву, яка починається з `a`[2]. У самій `f`, оголошення параметрів може звучати як

```
f(int arr[]) { ... }
```

або

```
f(int *arr) { ... }
```

Тож, що стосується `f`, сам факт того, що параметр посилається на частину більшого масиву не має значення.

Якщо впевнені, що елементи існують, можливо також вказати індекс у зворотньому напрямку у масиві: `p[-1]`, `p[-2]` і так далі. Це синтаксично допустимо, і вказує на елементи попереду `p[0]`. Звичайно, забороняється посилатися на об'єкти поза межами масиву.

5.4 Арифметика адрес

Маючи `p` як покажчик на певний елемент масиву, `p++` здійснить приріст `p`, тож він вказуватиме на наступний елемент, тоді як `p+=n` збільшить його настільки, що він вказуватиме на `n` елементів далі того місця, на яке посилається в дану мить. Ці і подібні конструкції є найпростішою формою адресної або покажчикової арифметики.

C є послідовною і систематичною стосовно свого підходу до арифметики адрес; інтеграція покажчиків, масивів і адресної арифметики є однією із сильних сторін мови. Проілюструймо це шляхом написання рудиментарного розподільника пам'яті. Ми створимо дві функції. Перша, `alloc(n)`, повертає покажчик на `n`-нне послідовне положення знака, яке викликова функція `alloc` може використати для збереження символів. Наступна, `afree(p)`, звільняє отримане місце зберігання, тож його можна використати пізніше. Ці рутини «рудиментарні», оскільки виклик `afree` потрібно здійснювати у протилежній послідовності від викликів `alloc`. Тобто, місце збереження, яким керують `alloc` із `afree`, є стеком, або ж «останній всередину — перший назовні». Стандартна бібліотека передбачає аналогічну функцію під назвою `malloc` і `free`, на котрі не накладено таких обмежень; у Розділі 8.7 ми покажемо, як їх можна реалізувати.

Найлегше втілення — змусити `alloc` надати, розбитий на частини, великий символьний масив, який ми назвемо `allocbuf`. Цей масив буде приватним для `alloc` і `afree`. Оскільки функції мають справу з покажчиками, а не індексами масивів, жодна інша функція не повинна знати назву масиву, який можна оголосити статичним, `static`, у вихідному файлі, в якому містяться `alloc` з `afree` — це зробить його невидимим ззовні. У реальному житті, масив може навіть не мати назви, його можна було би отримати викликом `malloc` або спитавши операційну систему надати покажчик на безіменний блок пам'яті.

Інша важлива інформація, це скільки саме `allocbuf` зайнято. Ми використали покажчик під назвою `allocp`, який вказує на наступний вільний елемент. Коли `alloc`

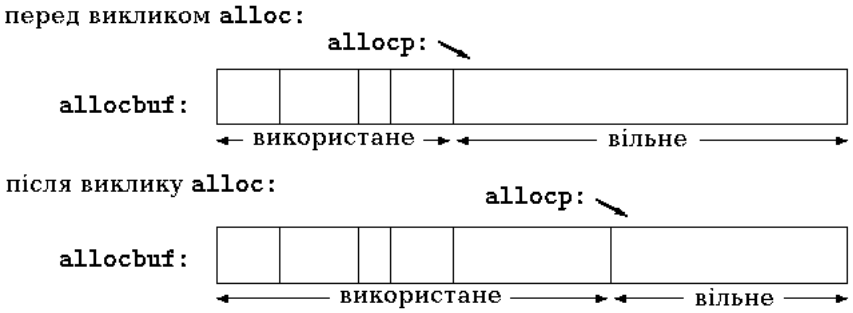
запитують розмістити n знаків, вона перевіряє, чи залишилось достатньо місця в allocbuf. Якщо так, alloc повертає поточне значення allocp (тобто початок вільного місця), після чого збільшує його на n, для вказівки на наступну вільну ділянку. Якщо ж місця не залишилось, alloc поверне нуль. Функція afree(p) всього-навсього встановлює allocp у p, якщо p знаходиться всередині allocbuf.

```
#define ALLOCSIZE 10000             /* розмір наявного місця */

static char allocbuf[ALLOCSIZE];    /* пам'ять для alloc */
static char *allocp = allocbuf;     /* наступна вільна позиція */

char *alloc(int n)            /* повертає покажчик на n знаків */
{
    if (allocbuf + ALLOCSIZE - allocp >= n) { /* чи вміщається */
        allocp += n;
        return allocp - n;          /* старий покажчик */
    } else                          /* якщо недостатньо місця */
        return 0;
}

void afree(char *p)      /* звільняє місце, на яке вказує p */
{
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}
```



Загалом, покажчик можна ініціювати так само, як і будь-яку іншу змінну, але, як правило, єдині значення, які мають сенс такому разі, це нуль або ж вираз, що включає адресу попередньо-визначених даних відповідного типу. Оголошення

```
static char *allocp = allocbuf;
```

визначає `allocp` як символьний покажчик і ініціалізує його як вказівник на початок `allocbuf`, тобто наступної вільної позиції під час запуску програми. Це можна також було би написати як

```
static char *allocp = &allocbuf[0];
```

оскільки назва масиву — це адреса елемента з індексом нуль. Умова

```
if (allocbuf + ALLOCSIZE - allocp >= n) {  /* чи вміщається */
```

перевіряє, чи достатньо місця для запиту на `n` знаків. Якщо так, нове значення `allocp` буде щонайбільше на одну позицію за кінцем `allocbuf`. Якщо запит можна задовольнити, `alloc` поверне покажчик на початок блока символів (зверніть увагу на оголошення самої функції). Якщо ні, `alloc` повинна повернути якийсь сигнал, що не залишилось більше місця. С гарантує, що нуль ніколи не буде чинною адресою для даних, тож нуль, як повернене значення, може бути використано для сигналізування анормального явища, в цьому випадку — відсутності вільного місця.

Покажчики і цілі — не взаємозамінні. Єдине виключення: нуль. Константу нуль можна присвоїти покажчику, і сам покажчик можна порівнювати з константою-нулем. Часто замість нуля використовується символічна константа `NULL`, просто як мнемоніка, щоб підкреслити, що це спеціальне значення для покажчика. `NULL` означено в <`stdio.h`>. Надалі, ми також вживатимемо `NULL`.

Перевірки на зразок

```
if (allocbuf + ALLOCSIZE - allocp >= n) {  /* вміщається */
```

та

```
if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
```

демонструють декілька важливих рис арифметики покажчиків. Перш за все, покажчики можна порівнювати за певних умов. Якщо `p` із `q` вказують на члени того самого масиву, тоді порівнювання на кшталт `==`, `!=`, `<`, `>=` тощо працюють як слід. Наприклад,

```
p < q
```

справджується, якщо `p` вказує на попередній до `q` елемент. Будь-який покажчик можна порівнювати на рівність чи не-рівність нулю. Але поводження буде невизначеним, якщо порівнювати покажчики, які не є членами того самого масиву. (Існує один виняток: адреса першого елемента за межами масиву, яку теж можна використати в покажчиковій арифметиці.)

Друге, на що ми звернули увагу, що покажчики та цілі можна додавати та віднімати. Конструкція

```
p + n
```

означає «адреса n-ного відносно того, на який зараз вказує p». Це є чинним, незалежно від типу об'єкту на який вказує p; n збільшується в масштабі, згідно розмірові об'єкту на який вказує p, що, в свою чергу, визначається оголошенням p. Якщо int дорівнює чотирьом байтам, наприклад, int буде збільшено в масштабі в чотири рази.

Чинним є також віднімання покажчиків; якщо p та q вказують на елементи того самого масиву і p<q, тоді q-p+1 дорівнює кількості елементів від p до q, включно. Цей факт можна використати для написання іншої версії strlen:

```
/* strlen:  повертає довжину ланцюжка s */
int strlen(char *s)
{
    char *p = s;

    while (*p != '\0')
        p++;
    return p - s;
}
```

В оголошенні, p ініційовано як s, тобто вказує на перший символ ланцюжка. Всередині циклу while, кожний символ перевіряється по черзі доти, поки не буде знайдено '\0' в кінці. Оскільки p вказує на символи, p++ просуває p до наступного символу кожного разу, а p-s дає кількість пройдених символів, себто довжину ланцюжка. (Кількість символів ланцюжка може бути завеликою для збереження в int. Заголовок <stddef.h> визначає тип ptrdiff_t, достатньо великий для збереження різниці значень двох покажчиків у вигляді числа зі знаком. Якщо бути ще обережнішим, можна використати size_t для значення, повернутого strlen, щоб збігалося із версією зі стандартної бібліотеки. Тип size_t є беззнаковим цілим, яке повертає оператор sizeof.

Арифметика покажчиків є досить послідовною: якщо би ми мали справу із числами з рухомою точкою (float), що займають більше місця ніж char, і якби p було покажчиком на таке число із рухомою точкою, p++ також би просунулось до наступного числа із рухомою точкою. Таким чином, ми могли би написати іншу версію alloc, яка би підтримувала числа із рухомою точкою замість символів (char), просто замінюючи char на float скрізь в alloc і afree. Всі операції з покажчиками автоматично беруть до уваги розмір об'єкту на який вказує покажчик.

Правильними операціями з покажчиками є присвоєння покажчиків того самого типу, додавання та віднімання покажчика та цілого, віднімання або порівняння двох покажчиків, що вказують на члени того самого масиву, а також присвоєння або порівнювання з нулем. Решта арифметичних дій з покажчиками є недійсною. Не дозволено

додавати два покажчики, або множити, ділити або порозрядно зміщувати чи маскувати їх, або ж додавати `float` чи `double` до них, або, навіть (за виключенням `void *`), присвоювати покажчик одного типу покажчику іншого без зведення типів.

5.5 Покажчики на символи та функції

Ланцюжкова константа, записана як

```
"I am a string"
```

являє собою масив знаків. У внутрішньому представленні, такий масив закінчується нульовим символом `'\0'`, щоб програми могли знайти кінець символьного масиву. Розмір для збереження, таким чином, буде на одиницю більшим за кількість знаків в лапках. Напевне, найчастіше ланцюжкові константи можна зустріти в якості аргументів функцій, як от

```
printf("hello, world\n");
```

Коли ланцюжок знаків як цей з'являється в програмі, доступ до нього відбувається через покажчик на символ; `printf` отримує покажчик на початок символьного масиву. Іншими словами, доступ до ланцюжкової константи відбувається через покажчик на її перший елемент.

Ланцюжкові константи не обов'язково є аргументами функцій. Якщо `pmessage` оголошено як

```
char *pmessage;
```

тоді твердження

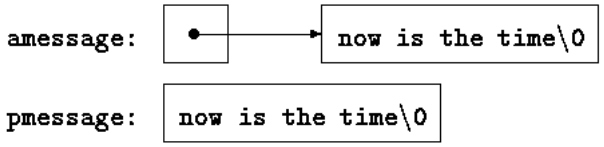
```
pmessage = "now is the time";
```

присвоює `pmessage` покажчик на символьний масив. Тут не відбувається жодного копіювання ланцюжків; задіяні лишень покажчики. C не передбачає жодних операторів для обробки цілого ланцюжка символів як єдиного цілого.

Існує одна важлива різниця між цими двома визначеннями:

```
char amessage[] = "now is the time"; /* масив */
char *pmessage = "now is the time"; /* покажчик */
```

Тут, `amessage` — це масив, досить великий, щоб втримати послідовність знаків і кінцевий `'\0'`, які йому присвоєно. Можна змінювати окремі знаки всередині масиву, але `amessage` завжди вказуватиме на те саме місце збереження в пам'яті. На противагу, `pmessage` — це покажчик, ініційований для того, щоб вказувати на ланцюжкову константу; покажчик згодом можна змінити так, щоб він вказував на інше місце, але ви отримаєте невизначений результат, якщо спробуєте змінити зміст самого ланцюжка.



Ми проілюструємо ще деякі риси покажчиків і масивів шляхом вивчення версій двох корисних функцій, адаптованих зі стандартної бібліотеки. Першою функцією є `strcpy(s,t)`, яка копіює ланцюжок `t` до ланцюжка `s`. Було би простіше, якби можна було написати просто `s=t`, але це копіює покажчик, а не символи. Для копіювання знаків нам потрібний цикл. Спершу, версія з масивом:

```
/* strcpy:    копіює t до s; версія з індексованим масивом */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}
```

Для порівняння, тут таки подаємо версію з покажчиками:

```
/* strcpy:    копіює t до s; версія з покажчиками */
void strcpy(char *s, char *t)
{
    int i;

    i = 0;
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

Позаяк аргументи передаються за значенням, `strcpy` може використовувати параметри `s` із `t` як їй заманеться. Тут це зручно ініційовані покажчики, якими проходяться по масиву, кожен символ по-черзі доти, доки `'\0'`, який завершує `t`, не скопійовано до `s`. На практиці, `strcpy` не було би написано так, як ми от це показали. Досвідчені С-програмісти надали би перевагу наступному

```
/* strcpy:    копіює t до s; друга версія з покажчиками */
void strcpy(char *s, char *t)
{
    while ((*s++ = *t++) != '\0')
        ;
}
```

У цьому варіанті, приріст `s` із `t` перенесено в тестову частину циклу. Значення `*t++` складається з символу, на який вказував `t` до того як відбувся приріст; постфіксний `++` не змінює `t` доти, доки не здобуто символ. Аналогічно, символ буде збережено до старого положення `s` до того як збільшити `s`. Цей символ одночасно є значенням, яке прирівнюється до `'\0'` для контролю над циклом. В кінцевому результаті, символи скопійовано з `t` до `s` аж до завершального `'\0'`, включно.

Як остаточне скорочення, зверніть увагу, що порівнювання з `'\0'` насправді — зайве, оскільки питання тільки в тому, чи є вираз нульовим. Тож функцію, скоріш за все, було би написано як

```
/* strcpy:    копіює t до s; третя версія з покажчиками */
void strcpy(char *s, char *t)
{
    while (*s++ = *t++)
        ;
}
```

Хоч це може видатись зашифрованим, з першого погляду, важливою є зручність нотації і цю ідіому варто засвоїти, ослільки ви часто бачитимете її в С-програмах. Функція `strcpy` зі стандартної бібліотеки (<`string.h`>) повертає кінцевий ланцюжок як значення функції.

Другою функцією, яку ми розглянемо, є `strcmp(s,t)`, яка порівнює символьний ланцюжок `s` із `t` і повертає від'ємне, нуль або додатнє значення, якщо лексикографічно `s` менший, рівний або більший за `t`. Результат буде отримано шляхом віднімання знаків у першому ж положенні, де `s` із `t` не узгоджуються.

```
/* strcmp:  повертає <0, якщо s<t; 0, якщо s==t; >0, якщо s>t */
int strcmp(char *s, char *t)
{
    int i;
```

```
    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;
    return s[i] - t[i];
}
```

Версія `strcmp` із використанням покажчиків:

```
/* strcmp:  повертає <0, якщо s<t; 0, якщо s==t; >0, якщо s>t */
int strcmp(char *s, char *t)
{
    for ( ; *s == *t; s++, t++)
        if (*s == '\0')
            return 0;
   return *s - *t;
}
```

Оскільки `++` та `-` — це або префіксні або постфіксні оператори, то можуть існувати також й інші комбінації з `*` або `++` або `-`, хоч і не так часто. Наприклад,

```
*--p
```

зменшує `p` до того як отримати символ, на який вказує `p`. Насправді, набір виразів

```
*p++ = val;      /* проштовхнути val у стек */
val = *--p;      /* виштовхнути верхівку стеку і присвоїти *
                  * це значення val */
```

— це стандартні ідіоми проштовхування і виштовхування зі стеку; подивіться Розділ 4.3.

Файл заголовка <`string.h`> містить оголошення функцій, згаданих у цьому розділі, так само різноманітних інших стандартної бібліотеки по обробці ланцюжків.

Вправа 5-3. Напишіть версію з покажчиками функції `strcat`, яку ми пройшли в Розділі 2; `strcat(s,t)` копіює ланцюжок `t` в кінець `s`.

Вправа 5-4. Напишіть функцію `strend(s,t)`, яка повертає 1, якщо ланцюжок `t` знайдено в кінці ланцюжка `s`, і нуль — якщо ні.

Вправа 5-5. Напишіть версії бібліотечних функцій `strncpy`, `strncat` і `strncmp`, які би діяли на щонайбільше `n` символів власних аргументів-ланцюжків. Наприклад, `strncpy(s,t,n)` копіює максимум `n` знаків `t` до `s`. Повні описи знаходяться в Додатку Б.

Вправа 5-6. Перепишіть наново відповідні програми та вправи з попередніх розділів, використовуючи покажчики замість індексації масивів. Хороші можливості складають `getline` (з розділів 1 і 4), `atoi`, `itoa` та їхні варіанти (Розділ 2, 3 та 4), `reverse` (Розділ 3) та `strindex` і `getop` (Розділ 4).

5.6 Масив покажчиків; покажчики на покажчики

Оскільки покажчики — це також змінні, їх можна зберегти в масивах, так само як і інші змінні. Дозвольте нам проілюструвати це шляхом написання програми, що сортує в алфавітному порядку набір рядків тексту. Це спрощена версія UNIX програми `sort`.

У Розділі 3, ми представили функцію сортування Шелла, яка сортує масив цілих чисел, і в Розділі 4, ми вдосконалили її за допомогою `quicksort`. Той самий алгоритм підійде і цього разу за виключенням того, що тепер нам доведеться мати справу з рядками тексту, які матимуть різну довжину і які, на противагу цілим числам, неможливо порівняти або перемістити тільки однією операцією. Нам знадобиться таке представлення даних, що успішно і легко справлятиметься з рядками тексту змінної довжини.

Саме тут вступають в гру масиви покажчиків. Якщо рядки, які треба посортувати, зберігати від початку до кінця в довгих символьних масивах, тоді до кожного рядка можна мати доступ через покажчик на його перший символ. Самі покажчики також можна зберегти в масиві. Два рядка можна порівняти, передаючи їх покажчики `strcmp`. Якщо ці два невпорядкованих рядки потрібно поміняти місцями, міняються місцями покажчики з масиву покажчиків, а не самі рядки тексту.

В такий спосіб ми можемо уникнути складної проблеми з керуванням збереженням даних і значної ресурсоємкості переміщення самих рядків.

Процес сортування складається з трьох етапів:

читання всіх рядків вводу
їхнє сортування
вивід впорядкованих рядків

Як звичайно, найкращим вирішенням буде поділити програму на функції, що збігаються з цим логічним поділом, функція `main` будучи керівною. Тимчасом, відкладемо етап сортування на потім, і зосередимось на структурі даних і вводі з виводом.

Функція вводу повинна одержати і зберегти символи кожного рядка і побудувати масив покажчиків до кожного. Вона також повинна порахувати кількість введених рядків, оскільки ця інформація знадобиться для сортування і виводу. Оскільки

функція вводу може обробити лише обмежену кількість введених рядків, вона може повернути недійсний рахунок, скажімо -1, якщо отримано забагато.

Функція виводу повинна тільки вивести рядки в тій послідовності, в якій вони розміщені в масиві покажчиків.

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

#define MAXLINES 5000       /* максимальна кількість рядків, що *
                             * буде відсортовано */
char *lineptr[MAXLINES];    /* покажчики на рядки тексту */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(char *lineptr[], int left, int right);

/* сортує введені рядки */
main()
{
    int nlines;            /* кількість прочитаних рядків вводу */

    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort(lineptr, 0, nlines-1);
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("error: input too big to sort\n");
        return 1;
    }
}

#define MAXLEN 1000      /* максимальна довжина будь-якого рядка */

int getline(char *, int);
char *alloc(int);

/* readlines:    читає введені рядки */
int readlines(char *lineptr[], int maxlines)
{
    int len, nlines;
    char *p, line[MAXLEN];
```

```
    nlines = 0;
    while ((len = getline(line, MAXLEN)) > 0)
        if (nlines >= maxlines || p = alloc(len) == NULL)
            return -1;
        else {
           line[len-1] = '\0';    /* вилучає символ нового рядка */
           strcpy(p, line);
           lineptr[nlines++] = p;
        }
    return nlines;
}

/* writelines:    виводить рядки */
void writelines(char *lineptr[], int nlines)
{
    int i;

    for (i = 0; i < nlines; i++)
        printf("%s\n", lineptr[i]);
}
```

Функцію `getline` ми запозичили з Розділу 1.9.
Новим для нас є оголошення `lineptr`:

```
char *lineptr[MAXLINES]
```

яке вказує на те, що `lineptr` є масивом з `MAXLINES` елементами, кожний з яких будучи покажчиком на `char`. Тобто, `lineptr[i]` — це покажчик на символ, а `*lineptr[i]` — це самий символ, на який він вказує, перший символ `i`-ного збереженого текстового рядка.

Оскільки `lineptr`, самий по собі, це назва масиву, його можна трактувати як покажчик, так само як і в попередніх прикладах, а `writelines` написати натомість як

```
/* writelines:    виводить рядки */
void writelines(char *lineptr[], int nlines)
{
    while (nlines-- > 0)
        printf("%s\n", *lineptr++);
}
```

Початково, `*lineptr` вказує на перший рядок; кожний елемент просуває його до наступного покажчика на рядок, у той час як `nlines` відраховує по спадній.

Маючи ввід і вивід під контролем, ми можемо перейти до сортування. Функція `quicksort` з Розділу 4 вимагає невеликих поправок: потрібно змінити оголошення і операцію порівнювання слід здійснювати за рахунок виклику `strcmp`. Сам алгоритм залишається тим самим, що додає нам певності, що це надалі працюватиме.

```
/* qsort: сортує v[left]...v[right] в порядку зростання */
   void qsort(char *v[], int left, int right)
{
    int i, last;
    void swap(char *v[], int i, int j);

    if (left >= right)   /* жодної дії, якщо масив містить */
        return;          /* менше ніж два елементи */
    swap(v, left, (left + right)/2);

    last = left;
    for (i = left+1; i <= right; i++)
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

Аналогічно, функція `swap` вимагає тільки незначних змін:

```
/* swap:    міняє місцями v[i] з v[j] */
void swap(char *v[], int i, int j)
{
    char *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Оскільки кожний окремий елемент `v` (синонім `lineptr`) — це покажчик на символ, змінна `temp` повинна бути такою самою, тож їх можна копіювати одне до одного.

Вправа 5-7. Перепишіть наново `readlines`, щоб вона зберігала рядки в масиві, наданому функцією `main`, замість виклику `alloc` для підтримки місця для зберігання. Наскільки швидшою буде програма?

5.7 Багатовимірні масиви

С надає прямокутні багатовимірні масиви, хоча на практиці вони набагато рідше вживані ніж масиви покажчиків. В цьому розділі ми продемонструємо деякі з їхніх властивостей.

Уявіть собі задачу по перетворенню дати з дня місяця в день року і навпаки. Наприклад, 1-го березня — це 60-ий день звичайного року і 61-ий день високосного. Давайте визначимо дві функції для здійснення перетворення: `day_of_year` перекладає місяць і день у день року, і `month_day`, яка переводить день року в місяць і день. Позаяк остання функція обчислює два значення, аргументи місяця і дня будуть покажчиками:

```
month_day(1988, 60, &m, &d)
```

присвоює `m` значення 2, а `d` значення 29 (29-го лютого).

Обидві ці функції потребують тієї самої інформації — таблиці з кількістю днів у кожному місяці («рівно 30 днів у вересні ...»). Оскільки кількість днів у високосному і невисокосному році відрізняється, то буде легше розділити їх на два ряди у двовимірному масиві, після чого звертати увагу на лютий під час обчислень. Масив і функції для здійснення перетворень слідують:

```
static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

/* day_of_year:    отримати день року маючи місяць і день */
int day_of_year(int year, int month, int day)
{
    int i, leap;
    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; i < month; i++)
        day += daytab[leap][i];
    return day;
}

/* month_day:    отримати місяць і день з дня року */
void month_day(int year, int yearday, int *pmonth,
               int *pday)

{
    int i, leap;
```

```
    leap = year%4 == 0 && year%100 != 0 || year%400 == 0;
    for (i = 1; yearday > daytab[leap][i]; i++)
            yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;
}
```

Якщо пригадуєте, арифметичним значенням логічного виразу, як у випадку `leap` (англ. високосний), може бути або нуль (хибно), або один (істина), тож це значення можна також використати як індекс масиву `daytab`.

Масив `daytab` повинен бути зовнішнім для обох, `day_of_year` і `month_day`, щоб обидва могли користуватись ним. Ми оголосили масив як `char` для ілюстрації того як використовувати `char` для збереження малих незнакових цілих.

`daytab` — це перший двовимірний масив з яким ми досі стикалися. У C, двовимірний масив, це насправді одновимірний, кожен елемент якого також є масивом. Тому індекси записуються як

```
daytab[i][j]          /* [рядок][стовпчик] */
```

замість

```
daytab[i,j]           /* НЕПРАВИЛЬНО */
```

За винятком цієї різниці в нотації, двовимірні масиви можна трактувати майже так само як і в інших мовах. Елементи зберігаються рядами, тож індекс з правого боку (або стовпчик), змінюється найшвидше, коли до елементів звертаються в послідовності, в які їх збережено.

Такий масив ініціалізується за допомогою, включеного у фігурні дужки, списку ініціалізаторів; кожний рядок двовимірного масиву ініціалізується відповідним другорядним списком. Ми почали масив `daytab` зі стовпчика, який містить 0, тож цифри місяців можна відраховувати природнішим 1 до 12, замість 0 до 11. Оскільки неможливо використати пропуск у цій ситуації, це зрозуміліше ніж пізніша зміна індексів.

При передачі двовимірного масиву як аргумент функції, оголошення параметра функції повинно включати кількість стовпчиків; кількість рядків не є настільки важливою, оскільки те, що передається, як ми раніше засвоїли, це покажчик на масив рядків, кожен з яких містить масив з 13-ти елементів типу `int`. У цьому конкретному випадку це покажчик на об'єкти, що являють собою масиви з 13-и `int`. Таким чином, якщо масив `daytab` передати функції `f`, оголошенням `f` буде:

```
f(int daytab[2][13]) { ... }
```

Також це могло би бути

```
f(int daytab[][13]) { ... }
```

оскільки кількість рядків не є важливою, або це могло б також бути

```
f(int (*daytab)[13]) { ... }
```

що вказує на те, що параметр — це покажчик на масив з 13-и цілих. Дужки потрібні, оскільки квадратні дужки [] мають більший пріоритет за `*`. Без дужок оголошення

```
int *daytab[13]
```

буде масивом з 13 покажчиків на цілі `int`. Як узагальнення, тільки перший вимір (індекс) масиву є вільним, решту потрібно вказувати. Розділ 5.12 включає подальше обговорення складних оголошень.

Вправа 5-8. Перевірки на помилки в `day_of_year` і `month_day` бракує. Виправіть цей недолік.

5.8 Ініціалізація масиву покажчиків

Уявіть собі завдання по написанню функції `month_name(n)`, яка би повертала покажчик на символьний ланцюжок, який містить назву `n`-нного місяця. Це являє собою ідеальне застосування внутрішнього статичного масиву. `month_name` включатиме приватний масив символьних ланцюжків, і повертатиме покажчик на відповідний, коли викликати її. Цей розділ демонструє, як ініціалізувати такий масив назв.

Синтаксис є подібним до попередніх ініціалізацій:

```
/* month_name:  повертає назву n-нного місяця */
char *month_name(int n)
{
    static char *name[] = {
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };

    return (n < 1 || n > 12) ? name[0] : name[n];
}
```

Оголошення `name` є масивом символьних покажчиків, таким самим як `lineptr` у прикладі з сортуванням. Ініціалізатором служить список символьних ланцюжків; кожному з них призначено відповідне положення в масиві. Символи `i`-ного ланцюжка розміщено в певному місці і покажчик на них збережено в `name[i]`. Оскільки розмір масиву name не вказано, компілятор сам порахує ініціалізатори і заповнить розмір правильним числом.

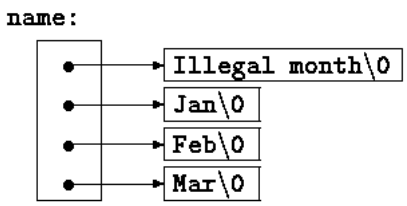
5.9 Покажчики в порівнянні з багатовимірними масивами

Новачки С, часом, плутаються з відмінністю між двовимірним масивом і масивом покажчиків, такому як `name`, у вищенаведеному прикладі. Маючи два визначення

```
int a[10][20];
int *b[10];
```

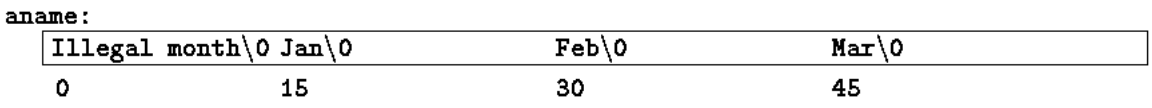
`a[3][4]` і `b[3][4]` обидва є синтаксично-правильними посиланнями на певну одиницю. Але `a` — це справжній двовимірний масив: під нього відведено 200 положень розміру `int`, і використовується традиційне прямокутне обчислення індексів `20*row+col` для того, щоб віднайти елемент `a[row,col]`. Проте, у випадку з `b`, таке визначення виділяє тільки 10 покажчиків, не ініціалізуючи їх; ініціалізацію потрібно здійснити явно — або статично, або ж за допомогою коду. Припускаючи, що кожний елемент `b` вказує на масив з двадцяти елементів, тоді буде відведено місце під 200 `int`, плюс десять комірок для покажчиків. Важливою перевагою масиву покажчиків є те, що рядки масиву можуть бути різноманітного розміру. Тобто, кожний елемент `b` не обов'язково повинен вказувати на вектор з двадцяти елементів; деякі можуть показувати на два елементи, інші — на п'ятдесят, а деякі — на жоден. Хоч ми й обійшлися в цій дискусії цілими числами, найчастіше використання масиву покажчиків полягає в збереженні символьних ланцюжків різноманітної довжини, як у функції `month_name`. Порівняйте оголошення і малюнок для масиву покажчиків:

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar" };
```



з аналогічними для двовимірного масиву:

```
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar" };
```



Вправа 5-9. Перепишіть наново функції `day_of_year` і `month_day` із покажчиками замість індексів.

5.10 Аргументи командного рядка

В середовищах, що підтримують C, існує спосіб передачі аргументів командного рядка, або параметрів, програмі, коли вона починає своє виконання. Головну функцію `main` викликано з двома аргументами. Перший (звично, названий `argc`, як скорочення від «argument count» — відлік аргументів) вказує на кількість аргументів командного рядка з яким було викликано програму; другий (`argv`, «argument vector» — вектор аргументів) є покажчиком на масив символьних ланцюжків, що, власне, містить аргументи — кожний ланцюжок відповідає аргументові. Ми, звичайно, використаємо багаторівневі покажчиків для маніпуляції цими символьними ланцюжками. Найпростішою ілюстрацією є програма `echo`, яка відлунює аргументи командного рядка на одному рядкові, розділяючи їх пробілами. Тобто, команда

```
echo hello, world
```

виводить

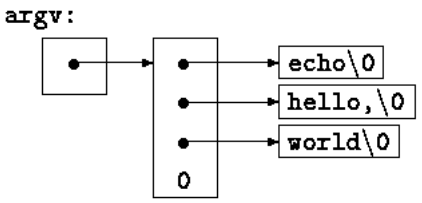
```
hello, world
```

За домовленістю, `argv[0]` — це назва, за якою було викликано програму, тож `argc` дорівнює, щонайменше, 1. Якщо `argc` дорівнює 1, то після назви програми на командному рядкові аргументів немає. У прикладі вище, `argc` дорівнює 3, тоді як `argv[0]`, `argv[1]` і `argv[2]` відповідають ¨`echo`¨, ¨`hello,`¨ та ¨`world`¨. Першим необов'язковим аргументом є `argv[1]`, а останнім — `argv[argc-1]`; на додачу, стандарт вимагає, щоб `argv[argc]` був нульовим покажчиком.

Наша перша версія `echo` розглядає `argv` як масив символьних покажчиків:

```
#include <stdio.h>

/* відлунює аргументи командного рядка; 1-а версія */
main(int argc, char *argv[])
```




```
{
    int i;

    for (i = 1; i < argc; i++)
        printf("%s%s", argv[i], (i < argc-1) ? " " : "");
    printf("\n");
    return 0;

}
```

Оскільки `argv` є покажчиком на масив, що складається з покажчиків, ми можемо маніпулювати покажчиками, замість індексами масиву. Наступний варіант основується на приростi `argv`, який є покажчиком на покажчик на `char`, одночасно здійснюється спад `argc`:

```
#include <stdio.h>

/* відлунює аргументи командного рядка; 2-а версія */
main(int argc, char *argv[])
{
    while (--argc > 0)
        printf("%s%s", *++argv, (argc > 1) ? " " : "");
    printf("\n");
    return 0;
}
```

Оскільки `argv`, це покажчик на початок масиву з ланцюжків аргументів, приріст його на одиницю (`++argv`) змусить його вказувати на `argv[1]` замість `argv[0]`. Кожний наступний приріст переносить його до наступного аргументу; `*argv`, таким чином, є покажчиком на цей аргумент. Одночасно, `argc` спадає; коли вона стає нулем, це означає, що аргументів для виводу на екран більше не залишилось.

Альтернативно, ми могли би написати вираз із `printf` як наступне:

```
printf((argc > 1) ? "%s " : "%s", *++argv);
```

Це демонструє, що формат аргументу `printf` може також бути виразом. Як ще один приклад, давайте покращимо програму знаходження за шаблоном із Розділу 4.1. Якщо ви пам'ятаєте, ми розмістили шаблон пошуку глибоко всередині програми — явно незадовільне розташування. Слідуючи прикладові UNIX-програми `grep`, давайте поліпшимо нашу, тож шаблон вказуватиметься як перший аргумент на командному рядкові.

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

/* find:   виводить рядки, що збіглися із шаблоном, вказаним
 * у 1-у аргументі */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    int found = 0;

    if (argc != 2)
        printf("Usage: find pattern\n");
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                printf("%s", line);
                found++;
            }
       return found;
}
```

Функція стандартної бібліотеки `strstr(s,t)` повертає покажчик на перший випадок ланцюжка `t` всередині ланцюжка `s`, або `NULL`, якщо жодного не знайдено. Її оголошено в <`string.h`>. Цю модель можна розвинути далі, для подальшої ілюстрації конструкцій з покажчиками. Скажімо, ми хотіли би дозволити два необов'язкових аргументи. Один вказуватиме «вивести всі рядки за винятком тих, що збігаються з шаблоном», а другий — «додати попереду кожного виведеного рядка його порядковий номер».

Загальною умовністю С-програм на UNIX-системах є те, що аргумент, який починається зі знака «мінус» означає необов'язковий прапорець або параметр. Якщо ми оберемо `-x` (як скорочення для «except») для вказівки протилежного результату, та `-n` («number»), для нумерації рядків, тоді команда

```
find -x -n шаблон
```

виведе кожний рядок, що не зійшовся з шаблоном, вказавши його порядковий номер. Необов'язкові аргументи треба дозволити у довільній послідовності і решта програми повинна бути незалежною від кількості аргументів, які ми надамо. Більше того, користувачам зручніше, якщо опції можна буде комбінувати, як от

```
find -nx шаблон
```

А ось і сама програма:

```
#include <stdio.h>
#include <string.h>
#define MAXLINE 1000

int getline(char *line, int max);

    /* find: виводить рядки, що збігаються із шаблоном,
     * вказаним у 1-му аргументові */
main(int argc, char *argv[])
{
    char line[MAXLINE];
    long lineno = 0;
    int c, except = 0, number = 0, found = 0;

    while (--argc > 0 && (*++argv)[0] == '-')
        while (c = *++argv[0])
            switch (c) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                printf("find: illegal option %c\n", c);
                argc = 0;
                found = -1;
                break;
            }
    if (argc != 1)
        printf("Usage: find -x -n pattern\n");
```

```
        else
            while (getline(line, MAXLINE) > 0) {
                lineno++;
                if ((strstr(line, *argv) != NULL) != except) {
                    if (number)
                        printf("%ld:", lineno);
                    printf("%s", line);
                    found++;
                }
            }
        return found;
    }
```

`argc` зменшено, а `argv` збільшено перед кожним аргументом. В кінці циклу, якщо не було помилок, `argc` вкаже, скільки аргументів залишилось необробленими і `argv` вказує на перший із них. Таким чином, `argc` повинен бути 1, а `*argv` повинен вказувати на шаблон. Зверніть увагу, що `*++argv` — це покажчик на ланцюжок аргументу, тому `(*++argv)[0]` — це перший його символ. (Альтернативною чинною формою є `**++argv`.) Через те, що [] зв'язуються тісніше за `*` із `++`, дужки обов'язкові; без них, вираз розглядався би як `*++(argv[0])`. Фактично, це те, що ми використали у внутрішньому циклові, де завдання полягає в проходженні через певний ланцюжок аргументу. У внутрішньому циклі, вираз `*++argv[0]` здійснює приріст покажчика `argv[0]`!

Рідко хто використовує вирази з покажчиками складніші за ці; в таких випадках, розбиття їй на два або три кроки буде інтуїтивнішим.

Вправа 5-10. Напишіть програму `expr`, яка обчислює зворотній польський запис на командному рядкові, де кожний оператор або операнд, це окремий аргумент. Наприклад,

```
expr 2 3 4 + *
```

обчислюється як `2 * (3+4)`.

Вправа 5-11. Змініть програму `entab` і `detab` (написані як вправи в Розділі 1), щоб вони могли взяти список табуляторних обмежувачів як аргумент. Використайте стандартні установлення табуляції, якщо аргументи відсутні.

Вправа 5-12. Розширте `entab` і `detab`, щоб вони дозволяти скорочення

```
entab -m +n
```

що означає табуляторний обмежувач кожні `n` стовпчиків, починаючи зі стовпчика `m`. Виберіть зручне (для користувача) уставне поводження.

Вправа 5-13. Напишіть програму `tail`, яка виводить останні `n` рядків свого вводу. Без задання, `n` буде задано як 10, скажімо, але його можна змінити за допомогою аргументу на командному рядку, тож

```
tail -n
```

виводить останні `n` рядків вводу. Програма повинна поводитись розумно, незалежно від того якими непередбачуваними є ввід і значення `n`. Напишіть цю програму таким чином, щоб вона найкраще використовувала місце зберігання; рядки повинні зберігатися як у програмі сортування з Розділу 5.6, а не в двовимірному масиві сталого розміру.

5.11 Покажчики на функції

У мові C, самі функції не є змінними, але існує можливість створення покажчиків на функції, які можна присвоїти, розмістити в масиві, передати іншим функціям, бути повернеными іншими функціями тощо. Ми проілюструємо це шляхом модифікації функції сортування, написаної раніше в цьому розділі, в такий спосіб, що якщо задано опцію `-n`, це сортуватиме ввідні рядки за числовим значенням замість лексикографічного.

Процес сортування часто складається з трьох частин: порівнювання — яке визначає порядок пар об'єктів, обмін — що змінює їхню послідовність, і алгоритм сортування — що здійснює порівнювання і обміни до тих пір, доки об'єкти не буде упорядковано. Алгоритм сортування повинен бути незалежним від операцій порівнювання і обміну, щоб через передачу йому різноманітних функцій порівнювання і обміну, ми могли добитися сортування за різними критеріями. Саме цей підхід застосовується у нашій новій програмі сортування.

Лексикографічне порівняння двох рядків, як і раніше, здійснюватиметься `strcmp`; нам також знадобиться нова функція `numcmp`, яка би порівнювала два рядки на основі числового значення і повертала такий самий вказівник стану, як це робить `strcmp`. Ці функції оголошено перед `main`, і покажчик до відповідної передаватиметься `qsort`. Ми знехтували обробкою помилок для аргументів, щоб зосередитись на основних питаннях.

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

#define MAXLINES 5000     /* максимальна кількість рядків для *
                           * сортування                      */
char *lineptr[MAXLINES];  /* покажчики на текст рядків */

int readlines(char *lineptr[], int nlines);
void writelines(char *lineptr[], int nlines);

void qsort(void *lineptr[], int left, int right,
           int (*comp)(void *, void *));
```

```
int numcmp(char *, char *);

/* сортує рядки вводу */
main(int argc, char *argv[])
{
    int nlines;          /* кількість прочитаних рядків вводу */
    int numeric = 0;     /* 1, якщо сортування за числовим    *
                          * значенням                         */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        qsort((void**) lineptr, 0, nlines-1,
              (int (*)(void*,void*))(numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    } else {
        printf("input too big to sort\n");
        return 1;
    }
}
```

У виклику `qsort`, `strcmp` і `numcmp` — це адреси функцій. Оскільки відомо, що це функції, оператор `&` необов'язковий, так само як його не потрібно перед назвою масиву.

Ми написали `qsort`, таким чином, що вона могла обробляти будь-який тип даних, а не тільки символьні ланцюжки. Як вказує прототипом функції, `qsort` очікує масиву покажчиків, два цілих і функцію з двома покажчиковими аргументами. Для останніх використовується загальний тип покажчика `void *`. Будь-який покажчик можна звести до `void *` і назад, без втрати інформації, тож ми можемо викликати `qsort` через зведення аргументів до `void *`. Складне зведення аргументів функції зводить також аргументи порівнювальної функції. Це, загалом, не матиме жодного ефекту на дійсному представленні, зате запевнить компілятор, що все гаразд.

```
/* qsort: сортує v[left]...v[right] у послідовності зростання */
void qsort(void *v[], int left, int right,
              int (*comp)(void *, void *))
{
    int i, last;

    void swap(void *v[], int, int);

    if (left >= right)  /* не робить нічого, якщо масив містить */
```

```
        return;         /* менше ніж два елементи                    */
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left+1; i <= right; i++)
        if ((*comp)(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1, comp);
    qsort(v, last+1, right, comp);
}
```

Оголошення варте вивчення. Четвертим параметром `qsort` є

```
int (*comp)(void *, void *)
```

яке вказує, що `comp` — це покажчик на функцію, яка має два аргументи типу `void *` і повертає `int`.

Використання `comp` у рядку

```
if ((*comp)(v[i], v[left]) < 0)
```

узгоджується з оголошенням — `comp` є покажчиком на функцію, `*comp` — це сама функція і

```
(*comp)(v[i], v[left])
```

— це її виклик. Дужки потрібні, щоб складові було приєднано належним чином; без них

```
int *comp(void *, void *)          /* НЕПРАВИЛЬНО */
```

вказує на те, що `comp` — це функція, яка повертає покажчик на `int`, що являється зовсім не тим, що потрібно. Ми вже продемонстрували функцію `strcmp`, яка порівнює два ланцюжки. Ось інша, `numcmp`, що порівнює ланцюжки за початковим числовим значенням, обчисленого викликом `atof`:

```
#include <stdlib.h>

/* numcmp:    порівнює s1 і s2 за числовим значенням */
int numcmp(char *s1, char *s2)
{
    double v1, v2;
```

```
    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    else if (v1 > v2)
        return 1;
    else
        return 0;
}
```

Функція `swap`, яка порівнює два покажчики, є тотожною тій, що ми представили раніше в цьому розділі за винятком того, що оголошення змінено на `void *`.

```
void swap(void *v[], int i, int j;)
{
    void *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

Можна також додати багато інших опцій до програми сортування, деякі можуть виявитись цікавими вправами.

Вправа 5-14. Змініть програму `sort` так, щоб вона брала прапорець `-r`, який вказував би сортування в оберненій (спадній) послідовності. Впевніться, що `-r` працює разом із `-n`.

Вправа 5-15. Додайте опцію `-f` для вирівнювання верхнього і нижнього регістрів, тож під час сортування не існуватиме регістрової відмінності; наприклад, `a` із `A` вважатимуться рівнозначними при порівнянні.

Вправа 5-16. Додайте опцію `-d` («directory order»), яка би здійснювала порівнювання тільки літер, чисел і пробілів. Впевніться, що вона працює разом із `-f`.

Вправа 5-17. Додайте можливість пошуку по ділянці, тож сортування можна здійснювати в окремих ділянках всередині рядків, кожне поле сортоване у відповідності з окремим набором опцій. (Індекс цієї книжки було посортовано за допомогою опцій `-df` для індексної категорії і опції `-n` для номерів сторінок.)

5.12 Складні оголошення

На мову C подеколи нарікають за синтаксис її оголошень, особливо тих, що стосуються покажчиків на функції. Цей синтаксис є спробою узгодити оголошення із використан-

ням; він добре працює в простих випадках, але може заплутати в складніших, позаяк оголошення не можна читати справа наліво і часом дужок занадто багато. Різниця між

```
int *f();     /* f: функція, що повертає покажчик на int */
```

і

```
int (*pf)();  /* pf: покажчик на функцію, що повертає int */
```

яскраво ілюструє цю проблему: `*` — це префіксний оператор і має нижчий пріоритет за `()`, тож дужки необхідні, щоб забезпечити відповідний зв'язок.

Хоча дійсно складні оголошення рідко зустрічаються на практиці, важливо знати як їх зрозуміти і, коли треба, як створити їх. Один з непоганих шляхів синтезувати оголошення, це за допомогою невеличких кроків із `typedef`, розглянутого в Розділі 6.7. Як альтернатива, в цьому розділі ми представимо пару програм, що перекладають з чинної C на людську мову, а потім назад. Словесний опис можна прочитати зліва направо.

Перша, `dcl`, дещо складніша. Вона перекладає оголошення C у слова, як, скажімо,

```
char **argv
    argv:    pointer to char
int (*daytab)[13]
    daytab:       pointer to array[13] of int
int *daytab[13]
    daytab:       array[13] of pointer to int
void *comp()
    comp: function returning pointer to void
void (*comp)()
    comp: pointer to function returning void
char (*(*x())[])()
    x: function returning pointer to array[] of
    pointer to function returning char
char (*(*x[3])())[5]
    x: array[3] of pointer to function returning
    pointer to array[5] of char
```

`dcl` основано на граматиці оголошень, яку точно описано в Додатку A, Розділі 8.5; ось спрощена форма:

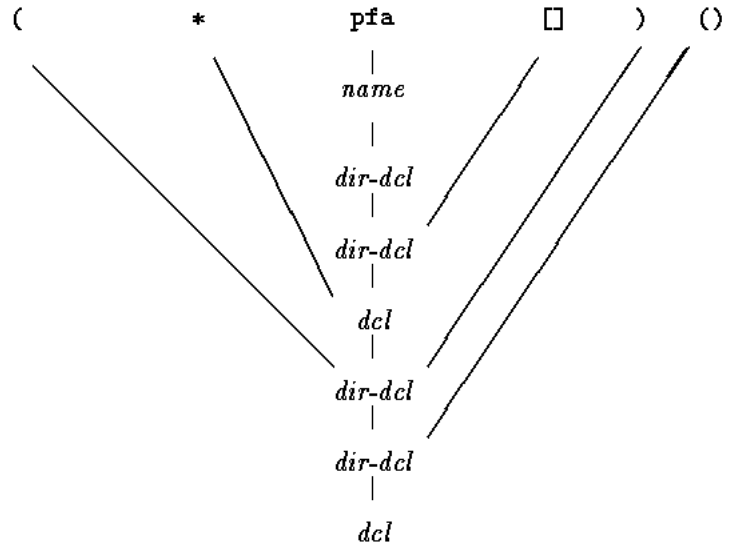
оголошувач: *необов'язковий(і)* * *прямий-оголошувач(і)*
прямий-оголошувач: *назва*
(*оголошувач*)
прямий-оголошувач()
прямий-оголошувач[*необов'язковий розмір*]

тобто, *оголошувач* — це *прямий-оголошувач*, можливо із передніми `*`. *прямий-оголошувач* — це або *назва*, або внесений в дужки *оголошувач*, або *прямий-оголошувач* із наступними дужками, або *прямий-оголошувач* з квадратними дужками і *необов'язковим розміром*.

Ця граматика може бути використана для аналізу оголошень. Наприклад, розглянемо оголошення

```
(*pfa[])()
```

де `pfa` буде ідентифіковано як *назву*, а тому як *прямий-оголошувач*. Тоді `pfa[]` — це також *прямий-оголошувач*. Після цього `*pfa[]` розпізнано як *оголошувач*, тож `(*pfa[])` є *прямим-оголошувачем*. Потім (*pfa[])() розпізнано як *прямий-оголошувач*, а тому як *оголошувач*. Ми можемо зобразити цей аналіз графічно як дерево (*прямий-оголошувач* скорочено до *пр-ог*):



Основою програми `dcl` є пара функцій — `dcl` і `dirdcl` — котрі обробляють оголошення відповідно до цієї граматики. Оскільки граматика означена рекурсивно, функції викликають одна одну рекурсивно по мірі того, як вони розпізнають частини оголошення; програма називається рекурсивно-спадним оброблювачем.

```
/* dcl:    прочитує оголошувач */
void dcl(void)
{
    int ns;

    for (ns = 0; gettoken() == '*'; ) /* count *'s */
          ns++;
    dirdcl();
    while (ns-- > 0)
          strcat(out, " pointer to");
}

/* dirdcl:    прочитує безпосередній оголошувач */
void dirdcl(void)
{
    int type;

    if (tokentype == '(') {           /* ( dcl ) */
          dcl();
          if (tokentype != ')')
               printf("error: missing )\n");
    } else if (tokentype == NAME)   /* variable name */
          strcpy(name, token);
    else
          printf("error: expected name or (dcl)\n");
    while ((type=gettoken()) == PARENS || type == BRACKETS)

             if (type == PARENS)
                  strcat(out, " function returning");
             else {
                  strcat(out, " array");
                  strcat(out, token);
                  strcat(out, " of");
             }
}
```

Ми хотіли, щоб програма була ілюстративною, а не куленепробивною, тому `dcl` багато в чому обмежена. Вона розуміє записи лише простих типів даних, таких як `char` або `int`. Вона не розуміє типи аргументів функцій або класифікатори на кшталт `const`. Випадкові пробіли заплутують її. Вона не здійснює якихось спроб обробки помилок, тож неправильні оголошення також заплутають її. Покращення цієї програми ми залишаємо вам як вправа.

Ось глобальні змінні і функція `main`:

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

#define MAXTOKEN          100

enum { NAME, PARENS, BRACKETS };

void dcl(void);
void dirdcl(void);

int gettoken(void);
int tokentype;            /* тип останньої лексеми */
char token[MAXTOKEN];     /* ланцюжок останньої лексеми */
char name[MAXTOKEN];      /* назва ідентифікатору */
char datatype[MAXTOKEN];  /* тип даних = char, int тощо */
char out[1000];

main()       /* перетворює оголошення на словесний опис */
{
     while (gettoken() != EOF) {       /* перша лексема рядка */
             strcpy(datatype, token);  /* is the datatype */
             out[0] = '\0';
             dcl();        /* читання решти рядка */
             if (tokentype != '\n')
                  printf("syntax error\n");
             printf("%s: %s %s\n", name, out, datatype);
     }
     return 0;
}
```

Функція `gettoken` опускає пробіли і табуляцію, після чого знаходить наступну лексему у вводі; лексемою може бути ім'я, пара дужок, пара квадратних дужок із можливим числом всередині, або будь-який інший одиничний знак.

```
int gettoken(void)    /* return next token */
{
    int c, getch(void);
    void ungetch(int);
    char *p = token;
```

```
        while ((c = getch()) == ' ' || c == '\t')
                ;
        if (c == '(') {
            if ((c = getch()) == ')') {
                strcpy(token, "()");
                return tokentype = PARENS;
            } else {
                ungetch(c);
                return tokentype = '(';
            }
        } else if (c == '[') {
            for (*p++ = c; (*p++ = getch()) != ']'; )
                ;
            *p = '\0';
            return tokentype = BRACKETS;
        } else if (isalpha(c)) {
            for (*p++ = c; isalnum(c = getch()); )
                *p++ = c;
            *p = '\0';
            ungetch(c);
            return tokentype = NAME;
        } else
            return tokentype = c;

    }
```

`getch` і `ungetch` обговорювались у Розділі 4.

Протилежний напрямок є легшим, особливо якщо ми не хвилюватимемось про створення зайвих дужок. Програма `undcl` перетворює словесний опис на зразок «x is a function returning a pointer to an array of pointers to functions returning char», який ми виразимо як

```
x () * [] * () char
```

на

```
char (*(*x())[])()
```

Скорочений синтаксис вводу дозволяє нам повторне використання функції `gettoken`. `undcl` також використовує ті самі зовнішні змінні що й `dcl`.

```
/* undcl: перетворює словесний опис на оголошення */
```

```
main()
{
    int type;
    char temp[MAXTOKEN];

    while (gettoken() != EOF) {
        strcpy(out, token);
        while ((type = gettoken()) != '\n')
            if (type == PARENS || type == BRACKETS)
                strcat(out, token);
            else if (type == '*') {
                sprintf(temp, "(*%s)", out);
                strcpy(out, temp);
            } else if (type == NAME) {
                sprintf(temp, "%s %s", token, out);
                strcpy(out, temp);
            } else
                printf("invalid input at %s\n", token);
            }
        return 0;
}
```

Вправа 5-18. Додайте перевірку на помилки і відновлення після помилок до `dcl`.

Вправа 5-19. Змініть `undcl` таким чином, щоб вона не додавала зайві дужки до оголошень.

Вправа 5-20. Розширте `dcl`, щоб вона обробляла оголошення з типами аргументів функцій, класифікатори на зразок `const` тощо.

Розділ 6

Структури

Структура — це набір з однієї або більше змінних, можливо різних типів, зібраних разом під одним ім'ям для зручного маніпулювання ними. (Структури також називають «записами» в інших мовах, а саме Pascal.) Структури допомагають організувати складні дані, особливо у великих програмах, оскільки вони дозволяють розглянути групу споріднених змінних як єдність, а не розрізнені одиниці.

Однією з типових структур є список зарплатні: опис працівника складається з набору ознак, таких як ім'я, адреса, номер соціального забезпечення, оклад тощо. Деякі з цих ознак, в свою чергу, теж могли би бути структурами: ім'я має декілька складових, те саме стосується адреси або навіть окладу. Інший, типовіший для C, приклад пов'язаний з графікою: пункт складається з пари координат, прямокутник — з пари пунктів і так далі.

Основною зміною, привнесеною стандартом ANSI, є опис присвоєння структурам — структури можна копіювати та надавати їм нових значень, їх може бути передано функціям і повернуто функціями. Ці риси підтримувались більшістю компіляторів впродовж багатьох років, але стандарт сформулював їх точніше. Автоматичні структури та масиви тепер також підтримуються.

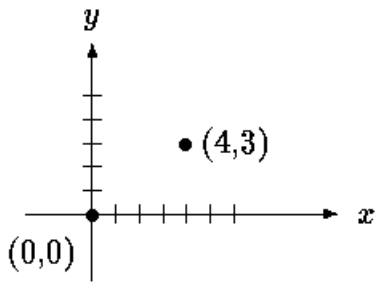
6.1 Основні поняття про структури

Створімо тепер декілька структур, придатних для графіки. Основним об'єктом є пункт, який має координати `x` і `y`, обидві цілі числа.

Ці дві складові можна помістити в структуру, оголошену як:

```
struct point {
      int x;
      int y;
};
```

Ключове слово `struct` оголошує структуру, яка в свою чергу є списком оголошень, включених у фігурні дужки. За словом `struct` може слідувати необов'язкова назва,



яку називають «етикеткою структури» (як `point` у цьому прикладі). Етикетка присвоює назву структурі, і може використовуватись пізніше як скорочення для частин, оголошених у фігурних дужках.

Змінні, вказані всередині, називаються елементами структури. Елемент структури, етикетка та звичайна, не включена в структуру, змінна можуть мати ту саму назву, не створюючи конфлікту, оскільки вони завжди розділені контекстом. Більше того, ті самі назви елементів можуть повторюватись у різних структурах, хоча, з міркувань стилю, таке використання назв можна застосувати хіба що для тісно взаємопов'язаних об'єктів.

Саме оголошення `struct` визначає новий тип. За правою закривною дужкою може слідувати список змінних, так само як і у випадку інших основних типів. Тобто, вираз

```
struct { ... } x, y, z;
```

синтаксично аналогічний

```
int x, y, z;
```

у тому сенсі, що обидва вирази оголошують змінні `x`, `y` і `z` як певного типу і зумовлюють відведення для них місця.

Оголошення структури, за яким не слідує список змінних, не зарезервує місця для зберігання; воно тільки описує зразок або форму структури. Однак, якщо оголошення включає етикетку, цю етикетку можна використати пізніше для означення окремих взірців структури. Так, наприклад, використовуючи вищенаведене оголошення `point`,

```
struct point pt;
```

визначить змінну `pt`, яка є структурою типу `struct point`. Структуру (змінну типу `struct`) можна започаткувати, якщо додати до її означення список ініціалізаторів, кожен з яких — це сталий вираз для окремих елементів структури:

```
struct point maxpt = { 320, 200 };
```

Автоматичну структуру також можна ініціювати через присвоєння або шляхом виклику функції, що повертає структуру правильного типу.

Елемент певної структури можна вказати в якомусь виразі конструкцією, що має форму

назва-структури.елемент

Оператор елемента структури «.» (крапка) зв'язує назву структури з одним з її елементів. Щоб вивести координати пункту `pt`, наприклад, нам потрібно вказати

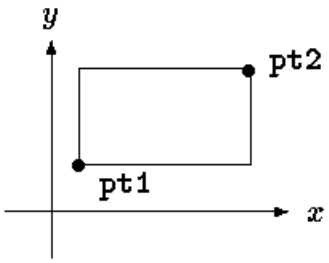
```
printf("%d,%d", pt.x, pt.y);
```

або, щоб обчислити відстань від початку координат (0,0) до `pt`,

```
double dist, sqrt(double);

dist = sqrt((double)pt.x * pt.x + (double)pt.y * pt.y);
```

Структури можуть гніздитися. Одним з представлень прямокутника може бути два пункти, що знаходяться у протилежних кутах:




```
struct rect {
    struct point pt1;
    struct point pt2;
};
```

Структура `rect` включає дві структури `point`. Якщо ми оголосимо `screen` як

```
struct rect screen;
```

тоді

```
screen.pt1.x
```

посилається на координату `x` елемента `pt1` структури `screen`.

6.2 Структури та функції

Єдиними чинними операціями зі структурами є їхнє копіювання, або присвоєння значення їм як цілому, здобуття їхньої адреси за допомогою &, та доступ до її членів. Копіювання та присвоєння включають передачу аргументів функціям так само як повернення значень функціями. Структури неможливо порівняти. Структуру можна ініціювати списком сталих значень членів структури; можна також започаткувати автоматичну структуру за допомогою присвоєння.

Давайте займемося дослідженням структур шляхом написання декількох функцій для роботи з пунктами прямокутника. Існує, принаймні, три можливих підходи до цієї проблеми: передача складових окремо, передача цілої структури, або передача покажчика на неї. Кожна з цих метод має свої переваги й недоліки.

Перша функція `makepoint` візьме в якості аргументів два цілих, і поверне структуру point:

```
/* makepoint: утворює пункт зі складників x та y */
struct point makepoint(int x, int y)
{
    struct point temp;

    temp.x = x;
    temp.y = y;
    return temp;
}
```

Зверніть увагу, що між назвою аргументу й елементом структури з тим самим ім'ям конфлікту не має; навпаки, повторне використання назви тільки підкреслює взаємозалежність.

`makepoint` тепер можна використати для динамічної ініціалізації структури, або подання структури як аргумент функції:

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepoint(0,0);
screen.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                        (screen.pt1.y + screen.pt2.y)/2);
```

Наступним кроком є створення набору функцій для арифметичних дій з пунктами. Наприклад

```
/* addpoints:  додає два пункти */
struct addpoint(struct point p1, struct point p2)
{
    p1.x += p2.x;
    p1.y += p2.y;
    return p1;
}
```

У цьому випадку, як аргументи, так і повернене значення функції являються структурами. Ми збільшили складові `p1` замість використання тимчасової змінної, щоб підкреслити, що структури, в якості параметрів, передаються за значенням, як і інші параметри функцій.

Як інший приклад, функція `ptinrect` перевіряє, чи пункт знаходиться всередині прямокутника, основуючись на нашій умові, що прямокутник включає ліву та нижню межу, а не верхню та праву:

```
/* ptinrect:  повертає 1, якщо p всередині r, 0 - якщо ні */
int ptinrect(struct point p, struct rect r)
{
    return p.x >= r.pt1.x && p.x < r.pt2.x
         && p.y >= r.pt1.y && p.y < r.pt2.y;
}
```

Це передбачає, що прямокутник представлено в стандартній формі, де координати `pt1` менші за координати `pt2`. Наступна функція повертає прямокутник, гарантовано в канонічній формі представлення:

```
#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))

/* canonrect:  стандартизує координати прямокутника */
struct rect canonrect(struct rect r)
{
    struct rect temp;

    temp.pt1.x = min(r.pt1.x, r.pt2.x);
    temp.pt1.y = min(r.pt1.y, r.pt2.y);
    temp.pt2.x = max(r.pt1.x, r.pt2.x);
    temp.pt2.y = max(r.pt1.y, r.pt2.y);
    return temp;
}
```

Якщо функції потрібно передати велику структуру, то загалом ефективніше вказати покажчик, чим копіювати цілу структуру. Покажчики на структуру подібні на покажчики на звичайні змінні. Оголошення

```
struct point *pp;
```

вказує на те, що `pp` — це покажчик на структуру типу `struct point`. Якщо `pp` вказує на структуру point, то `*pp` — це сама структура, а `(*pp).x` та `(*pp).y` — це члени структури. Для використання `pp`, ми могли би написати, наприклад,

```
struct point origin, *pp;
pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);
```

Дужки обов'язкові у випадку `(*pp).x`, оскільки пріоритет оператора елемента структури `.` більший за `*`. Вираз `*pp.x` означає `*(pp.x)`, що неправильно тому, що `x` не є покажчиком.

Покажчики на структури використовуються настільки часто, що було надано альтернативне позначення для скорочення. Якщо `p` — це покажчик на структуру, тоді

`p->`*член-структури*

посилається на певний елемент. Таким чином, ми могли би написати натомість

```
printf("origin is (%d,%d)\n", pp->x, pp->y);
```

Обидва, `.` та `->` спрягаються зліва направо, тож якщо ми матимемо

```
struct rect r, *rp = &r;
```

то наступні чотири вирази еквівалентні:

```
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

Структурні оператори `.` із `->`, разом із `()` виклику функцій і `[]` індексів знаходяться на верхівці ієрархії пріоритету, тож спрягаються дуже тісно. Отже, наприклад, якщо ми маємо оголошення

```
struct {
    int len;
    char *str;
} *p;
```

то

```
++p->len
```

здійснює приріст `len`, а не `p`, оскільки неявні дужки виглядають як `++(p->len)`. Дужки можна використати, щоб змінити спрягання (зв'язування): `(++p)->len` збільшує `p` перед тим як дістатися до `len`, тоді як `(p++)->len` збільшує `p` після. (Цей останній набір дужок необов'язковий.) Подібно до цього, `*p->str` добуває значення, на яке вказує `str`; `*p->str++` здійснює приріст `str` після доступу до того, на що вона вказує (схоже до `*s++`); `(*p->str)++` збільшує те, на що вказує `str`; а `*p++->str` збільшує `p` після доступу до того, на що вказує `str`.

6.3 Масиви структур

Уявімо собі, що нам треба написати програму, яка би рахувала кількість знайдених ключових слів C. Нам потрібен би був масив символьних ланцюжків, який би містив назви, і масив цілих для відліку. Одним з можливих варіантів було би використання двох паралельних масивів `keyword` і `keycount`, як от

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

Але самий факт того, що масиви паралельні, підказує відмінну організацію — масив структур. Кожне ключове слово складатиметься з пари:

```
char *word;
int cout;
```

і ці пари утворюватимуть масив. Оголошення структури

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

заявляє про структуру типу `key`, і означує масив `keytab`, що міститиме структури цього типу, відводячи для них місце в пам'яті. Кожен елемент масиву буде структурою. Це можна також написати як

```
struct key {
    char *word;
    int count;
};

struct key keytab[NKEYS];
```

Оскільки структура `keytab` містить сталий набір імен, найлегшим буде зробити її зовнішньою змінною й ініціювати раз і назавжди під час її означення. Ініціалізація структури аналогічна попереднім — за визначенням слідує список ініціалізаторів, включених у фігурні дужки:

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    "char", 0,
    "const", 0,
    "continue", 0,
    "default", 0,
     /* ... */
    "unsigned", 0,
    "void", 0,
    "volatile", 0,
    "while", 0
};
```

Ініціалізатори перелічено парами, згідно елементів структури. Було би точніше включити ініціалізатори кожного «рядка» структури у фігурні дужки, як от

```
{ "auto", 0 },
{ "break", 0 },
{ "case", 0 },
...
```

але внутрішні дужки — необов'язкові, якщо ініціалізатори складаються з простих змінних або символьних ланцюжків, і коли всі присутні. Зазвичай, кількість елементів масиву `keytab` обчислюється автоматично, при наявності ініціалізаторів і порожньому []. Програма відліку ключових слів почнеться з означення `keytab`. Функція `main` читатиме ввід шляхом повторного виклику `getword`, яка добуватиме по одному слову за раз. Кожне слово шукатиметься в `keytab` за допомогою нашої версії функції бінарного пошуку, написаної в Розділі 3. Список ключових слів потрібно буде сортувати в послідовності зростання в таблиці.

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

#define MAXWORD 100

int getword(char *, int);
int binsearch(char *, struct key *, int);

/* відлік ключових слів C */
main()
{
    int n;
    char word[MAXWORD];

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                keytab[n].count++;
    for (n = 0; n < NKEYS; n++)
        if (keytab[n].count > 0)
            printf("%4d %s\n",
                keytab[n].count, keytab[n].word);
    return 0;
}

/* binsearch:  знаходить слово word у tab[0]...tab[n-1] */
int binsearch(char *word, struct key tab[], int n)
{
    int cond;
    int low, high, mid;

    low = 0;
    high = n - 1;
```

```
        while (low <= high) {
            mid = (low+high) / 2;
            if ((cond = strcmp(word, tab[mid].word)) < 0)
                high = mid - 1;
            else if (cond > 0)
                low = mid + 1;
            else
                return mid;
        }
        return -1;
    }
```

Ми покажемо функцію `getword` за якусь мить; поки-що досить знати, що кожний її виклик знаходить нове слово, яке скопійовано в масив, вказаний як її перший аргумент. Кількість `NKEYS` є числом ключових слів у `keytab`. Хоча ми могли би самі порахувати це число, набагато легше і безпечніше залишити це машині, особливо якщо список може змінитися пізніше. Однією з можливостей буде завершити список ініціалізаторів нульовим покажчиком, після чого ітерувати через `keytab` до досягнення кінця.

Але це вкрай потрібно, оскільки розмір масиву повністю визначається під час компіляції. Розмір масиву складає розмір одного елемента помножено на кількість елементів, тож кількість складатиме просто

розмір `keytab` / *розмір* `struct key`

C забезпечує унарним компіляційним оператором під назвою `sizeof`, який можна застосувати для визначення розміру будь-якого об'єкту. Вирази

`sizeof` *об'єкт*

і

`sizeof (`*назва типу*`)`

повертають ціле число, рівне розмірові в байтах вказаного об'єкту або типу. (Точніше, `sizeof` повертає беззнакове ціле значення, чий тип `size_t` визначено в файлі заголовку <`stddef.h`>.) Об'єктом може служити як змінна, так і масив або структура. Назва типу також може бути однією з назв основних типів, таких як `int` або `doble`, або ж назвою похідного типу, як структура або покажчик. В нашому випадку, кількість ключових слів дорівнюватиме розмірові масиву, поділено на розмір одного елемента. Це обчислення використано в твердженні `#define` для того, щоб встановити значення `NKEYS`:

```
#define NKEYS (sizeof keytab / sizeof(struct key))
```

Іншим шляхом буде ділення масиву на розмір певного елемента:

```
#define NKEYS (sizeof keytab / sizeof(keytab[0]))
```

Останнє має перевагу в тому, що його не потрібно міняти, якщо тип змінено. `sizeof` неможливо використати в рядку `#if`, оскільки препроцесор не розуміє назв типів. Зате вираз у `#define` не обчислюється препроцесором, тож цей код чинний.

Тепер, щодо функції `getword`. Ми написали більш узагальнену `getword`, ніж ця програма потребує, але вона не є складною. `getword` добуває наступне «слово» зі вводу, де словом може служити або ланцюжок з літер і цифр, який починається з літери, або один символ, яке не є пробілом. Функція повертає перший знак слова, або `EOF` у випадку кінця файла, або самий знак, якщо він не є частиною алфавіту.

```
/* getword: одержує наступне слово зі вводу */
int getword(char *word, int lim)
{
    int c, getch(void);
    void ungetch(int);
    char *w = word;

    while (isspace(c = getch()))
        ;
    if (c != EOF)
        *w++ = c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for ( ; --lim > 0; w++)
        if (!isalnum(*w = getch())) {
            ungetch(*w);
            break;
        }
    *w = '\0';
    return word[0];
}
```

`getword` використовує `getch` і `ungetch`, які ми написали в Розділі 4. Коли набір буквенно-цифрових лексем закінчився, це означає, що `getword` зайшла на один символ задалеко. Виклик `ungetch` проштовхує символ назад на ввід для наступного виклику

функції. `getword` також використовує `isspace`, для ігнорування пробілів, `isalpha`, щоб розпізнати літери, і `isalnum` для розрізнення літер і цифр; усі зі стандартного файла заголовка <`ctype.h`>.

Вправа 6-1. Наша версія `getword` не обробляє належним чином жорсткі пробіли, ланцюжкові константи, коментарі, або вказівки препроцесору. Напишіть кращу версію.

6.4 Покажчики на структури

Для ілюстрації деяких міркувань щодо покажчиків на структури та масивів структур, напишімо нашу програму підрахунку ключових слів знову, цього разу використовуючи покажчики замість індексів масиву.

Зовнішнє оголошення `keytab` не потребує змін, зате `main` і `binsearch` — так.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAXWORD 100

int getword(char *, int);
struct key *binsearch(char *, struct key *, int);

/* лічить ключові слова C; версія з покажчиками */
main()
{
    char word[MAXWORD];
    struct key *p;

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p=binsearch(word, keytab, NKEYS)) != NULL)
                p->count++;
    for (p = keytab; p < keytab + NKEYS; p++)
        if (p->count > 0)
            printf("%4d %s\n", p->count, p->word);
    return 0;
}

/* binsearch: знаходить слово у tab[0]...tab[n-1] */
struct key *binsearch(char *word, struck key *tab, int n)
{
    int cond;
```

```
    struct key *low = &tab[0];
    struct key *high = &tab[n];
    struct key *mid;

    while (low < high) {
        mid = low + (high-low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }
    return NULL;
}
```

Тут існує декілька речей, вартих, щоб на них звернути увагу. Перш за все, оголошення `binsearch` повинно вказувати, що функція повертає покажчик на `struct key`, замість цілого; це оголошено в обох місцях, прототипові функції, та самій `binsearch`. Коли `binsearch` знаходить слово, вона повертає покажчик на нього; якщо зазнала невдачі — `NULL`.

Друге — елементи `keytab` тепер досяжні через покажчики. Це вимагає істотних змін `binsearch`.

Ініціалізатори `low` і `high` тепер покажчики на початок і поза кінець таблиці.

Обчислення середнього елемента тепер не може бути звичайним

```
mid = (low+high) / 2           /* НЕПРАВИЛЬНО */
```

оскільки додавання покажчиків заборонено. Віднімання, однак, дозволено, тож `high-low` дорівнюватиме кількості елементів, таким чином

```
mid = low + (high-low) / 2
```

встановить `mid` покажчиком на елемент посередині між `low` і `high`.

Найважливішою зміною є поправка алгоритму, щоб впевнитися, що він не видасть недійсного покажчика або намагатиметься дістатися до елемента поза межами масиву. Проблема в тому, що `&tab[-1]` і `&tab[n]`, обидва, знаходяться зовні масиву `tab`. Перший вираз просто недійсний, тоді як на другий не дозволено посилатися. Проте означення мови гарантує, що арифметика покажчиків, що стосується першого елемента поза межами масиву (тобто `tab[n]`) працюватиме правильно.

У `main` ми написали

```
for (p = keytab; p < keytab + NKEYS; p++)
```

Якщо `p`, це покажчик на структуру, то арифметика з `p` бере до уваги розмір структури, тож `p++` збільшує `p` на потрібне значення, щоб дістатися до наступного елемента масиву структур, а тестова умова зупиняє цикл у потрібну мить.

Не гадайте, однак, що розмір структури рівний сумі розмірів її членів. Через вимоги вирівнювання різноманітних об'єктів, можуть існувати безіменні «дірки» в структурі. Таким чином, наприклад, якщо розмір `char` рівний одному байтові, а `int` — чотирьом, структура

```
struct {
    char c;
    int i;
};
```

може, тим не менш, вимагати восьми байтів, а не п'яти. Оператор `sizeof` поверне справжнє значення. І, нарешті, відступ щодо формату програми: коли функція повертає складний тип на зразок покажчика на структуру, як от

```
struct key *binsearch(char *word, struct key *tab, int n)
```

назву функції може бути важко побачити або знайти в текстовому редакторі. Відповідно, часом використовується альтернативний стиль запису:

```
struct key *
binsearch(char *word, struct key *tab, int n)
```

Це залежить від особистих уподобань; виберіть ту форму, яка вам до вподоби та дотримуйтеся її.

6.5 Структури зі зворотнім звертанням

Скажімо, ми хотіли би справитися з більш узагальненою проблемою підрахунку всіх слів вводу. Оскільки список слів невідомий заздалегідь, нам не вдасться зручно посортувати та використати бінарний пошук. Одночасно, використання лінійного пошуку для порівняння кожного нового слова з попередніми не є ефективним; програма займе забагато часу. (Якщо точніше, то час обігу програми зросте в квадраті до кількості введених слів.) Як можна організувати дані для того, щоб можна було ефективно упоратись зі списком довільних слів?

Одним з виходів із даного становища є збереження сортованого набору зустрінутих слів через розміщення кожного слова одразу у відповідне положення під час

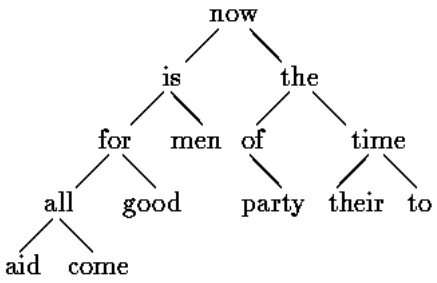
надходження. Однак, це не слід робити шляхом зміщення слів у лінійному масиві — це також забирає забагато часу. Натомість ми використаємо структуру даних під назвою бінарне дерево.

Дерево містить по одному «вузлу» на окреме слово; кожний вузол включає

- покажчик на текст слова,
- відлік випадків цього слова,
- покажчик на лівий дочірній вузол,
- покажчик на правий дочірній вузол.

Жоден вузол не може мати більш ніж двоє дочірніх вузлів; він може також мати нуль або один.

Вузли зберігаються в такий спосіб, що будь-який вузол у лівій частині відгалуження містить тільки слова, лексикографічно менші за слово вузла, а праві відгалуження — лексикографічно більші. Ось дерево для речення «now is the time for all good men to come to the aid of their party» побудоване шляхом додавання кожного слова під час його знаходження:



Щоб дізнатися, чи нове слово вже було внесено в дерево, розпочніть з кореня і порівняйте нове слово зі словом, збереженим у цьому вузлі. Якщо вони збіглися, відповідь є ствердувальною. Якщо нове слово менше за вузлове, шукайте далі в лівому дочірньому вузлі, у протилежному випадку — в правому. Якщо не залишилось дочірніх відгалужень у заданому напрямку, нове слово не було внесено в дерево, і дійсно, порожній сегмент і буде тим місцем, де потрібно зберегти нове слово. Цей процес є рекурсивним, оскільки пошук з будь-якого вузла розпочинає новий з одного зі своїх дочірніх. Відповідно, натуральнішою буде рекурсивна функція для додання і виводу.

Повертаючись назад до опису вузла, найзручнішим буде представити його як структуру з чотирьох складових:

```
struct tnode {              /* вузол дерева: */
    char *word;                  /* покажчик на ланцюжок */
    int count;                   /* кількість його повторень */
```

```
    struct tnode *left;         /* лівий дочірній вузол */
    struct tnode *right;        /* правий дочірній вузол */
};
```

Це рекурсивне оголошення вузла може виглядати складеним навмання, але воно правильне. Структурі не дозволяється включати копію самої себе, але

```
struct tnode *left;
```

оголошує `left` як покажчик на `tnode`, а не саму `tnode`.

Інколи, вам можуть знадобитися дві зворотньо-звернені структури: дві структури, що посилаються одна на одну. Щоб добитися цього, потрібно

```
struct t {
    ...
    struct s *p;        /* p є покажчиком на s */
};
struct s {
    ...
    struct t *q;        /* q є покажчиком на t */
};
```

Код самої програми — на дивовижу короткий, маючи жменю вже написаних нами допоміжних функцій, таких як `getword`. Функція `main` читає слова, користуючись `getword`, і розміщає їх у дереві за допомогою `addtree`.

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

#define MAXWORD 100
struct tnode *addtree(struct tnode *, char *);
void treeprint(struct tnode *);
int getword(char *, int);

/* word frequency count */
main()
{
    struct tnode *root;
    char word[MAXWORD];

    root = NULL;
```

```
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);
    return 0;
}
```

Наступна функція, `adtree`, — рекурсивна. Функція `main` розмістить перше слово на найвищому рівні (корені) дерева. На кожній стадії, нове слово порівнюватиметься зі вже збереженим вузловим словом, і буде переміщено вниз до правого або лівого відгалуження через рекурсивний виклик `adtree`. Зрештою, слово може збігтися з чимось, вже привнесеним у дерево (тоді просто збільшується відлік цього слова), або відбудеться зіткнення з нульовим покажчиком, що вказуватиме, що потрібно створити вузол і додати його до дерева. Коли створено новий вузол, `addtree` повертає покажчик на нього, що буде розміщено у батьківському вузлі.

```
struct tnode *talloc(void);
char *strdup(char *);

/* addtree:  додає вузол із w, в або нижче p */
struct tnode *addtree(struct tnode *p, char *w)
{
    int cond;

    if (p == NULL) {          /* надійшло нове слово */
        p = talloc();         /* створити новий вузол */
        p->word = strdup(w);
        p->count = 1;
        p->left = p->right = NULL;
    } else if ((cond = strcmp(w, p->word)) == 0)
        p->count++;     /* повторне слово */
    else if (cond < 0)  /* якщо менше - ліве відгалуження */
        p->left = addtree(p->left, w);
    else                /* якщо більше - праве відгалуження */
        p->right = addtree(p->right, w);
    return p;
}
```

Місце для зберігання нового вузла забезпечується функцією `talloc`, яка повертає покажчик на вільне місце, придатне для утримання вузла дерева, і нове слово копіюється до прихованого місця за допомогою `strdup`. (Ми розглянемо ці функції за якусь

мить.) Ініціалізується `count` (змінна відліку), два дочірніх вузли отримають значення `NULL`. Ця частина коду виконується тільки в «кроні» дерева, під час додання нового вузла. Ми (нерозумно) пропустили перевірку на помилки щодо значень повернутих `strdup` і `talloc`.

`treeprint` виводить дерево в сортованому вигляді; для кожного вузла, вона виводить ліве відгалуження (всі слова менші за дане), потім саме слово, потім праве відгалуження (всі слова більші за дане). Якщо ви почуваєтесь невпевнено, щодо того, як працює рекурсія, можете симулювати роботу `treeprint`, як показано нижче.

```
/* treeprint:  впорядкований вивід дерева p */
void treeprint(struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

Практичне зауваження: якщо дерево стає «незбалансованим» від того, що слова не надходять у довільній послідовності, час обігу програми може суттєво збільшитись. У найгіршому випадку, коли слова вже знаходяться в правильній послідовності, ця програма здійснить ресурсоємку симуляцію лінійного пошуку. Існують більш узагальнені версії бінарного дерева, що не страждають від цього найгіршого випадку, але ми не обговорюватимемо їх тут.

Перед тим як залишити цей приклад, варто зробити невеличкий відступ щодо розподільників пам'яті. Очевидно, що бажано мати тільки один розподільник у програмі, навіть якщо від розподіляє під різнорідні об'єкти. Але якщо існує тільки один розподільник для обробки запитів щодо, скажімо, покажчиків на `char`, і покажчики на `struct tnode`, тоді виникають два питання. Перш за все, як він виконуватиме вимогу більшості машин щодо того, що об'єкти певних типів повинні вдовольняти обмеження по вирівнюванню (наприклад, що цілі, часто, повинні знаходитись у парних адресах)? Друге, які оголошення можуть справитись із фактом того, що розподільник повинен обов'язково повернути різні типи покажчиків?

Вимоги по вирівнюванню, загалом, можна легко вдовольнити ціною деякого змарнованого простору, впевнившись, що розподільник завжди повертає покажчик, що відповідає всім обмеженням по вирівнюванню. Функція `alloc` з Розділу 5 не гарантує жодного вирівнювання, тож ми скористаємося з функції стандартної бібліотеки `malloc`, яка упорається з цим. У Розділі 8, ми покажемо один із способів втілення `malloc`.

Питання, щодо оголошення типу для таких функцій як `malloc`, є головною біллю для будь-якої мови, що піклується про перевірку типів. У C, чинним є оголосити,

що `malloc` повертає покажчик на `void`, що дозволяє перетворити покажчик на бажаний тип за допомогою зведення типу. `malloc` і споріднені функції оголошено в файлі заголовка <`stdlib.h`>. Тож, функцію `talloc` можна написати як

```
#include <stdlib.h>

/* talloc:  створює tnode */
struct tnode *talloc(void)
{
    return (struct tnode *) malloc(sizeof(struct tnode));
}
```

`strdup` просто копіює, наданий їй як аргумент, ланцюжок у безпечне місце, здобуте викликом `malloc`:

```
char *strdup(char *s)         /* копіює s */
{
    char *p;

    p = (char *) malloc(strlen(s)+1);  /* +1 для '\0' */
    if (p != NULL)
        strcpy(p, s);
    return p;
}
```

`malloc` повертає `NULL`, якщо не залишилось місця; `strdup` передає це значення далі, залишаючи обробку помилок тому, хто її викликав.

Місце, відведене `malloc` можна звільнити для перевикористання за допомогою `free`.

Вправа 6-2. Напишіть програму, що читатиме вихідний текст на C і виводитиме в алфавітній послідовності кожну групу назв змінних, перші 6 літер яких збігаються. Нехай вона не бере до уваги ланцюжки і коментарі. Зробіть 6 параметром, який можна встановити з командного рядка.

Вправа 6-3. Напишіть програму перехресного посилання, яка би виводила список усіх слів у документі, і для кожного слова, список номерів рядків, де воно з'являється. Не звертайте уваги на «шум», такі слова як «the», «for» тощо.

Вправа 6-4. Напишіть програму, яка би виводила окремі слова вводу, сортовані у послідовності спадання за кількістю повторень. Додайте число повторень кожного слова попереду.

6.6 Пошук по таблиці

У цьому розділі, ми напишемо осердя програми пошуку за таблицею, щоб проілюструвати додаткові риси структур. Цей код типовий для того, що можна знайти у функціях по обробці таблиці символів макро-процесору або компілятора. Наприклад, розглянемо твердження `#define`. Коли ми зустріли рядок на кшталт

```
#define     IN     1
```

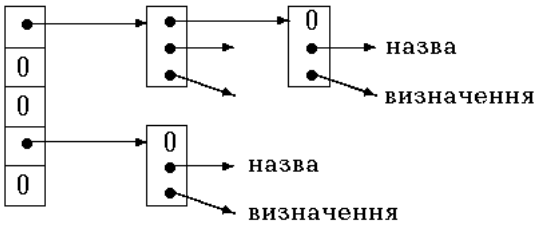
назва `IN` і її текст заміни `1` буде збережено в таблиці. Пізніше, коли `IN` з'явиться у виразі на зразок

```
state = IN;
```

її заступись `1`.

Ми побачимо дві функції, по обробці назв і текстів заміни. `install(s,t)` реєструє в таблиці назву `s` і текст заміни `t`; `s` із `t`, це просто символьні ланцюжки. `lookup(s)` шукає `s` у таблиці, і повертає покажчик на місце, де його знайдено, або `NULL`, як такого ланцюжка там немає.

Використовується алгоритм гешованого пошуку — новоприбуле ім'я перетворено на невелике додатнє ціле, яке пізніше використовується як індекс масиву покажчиків. Елемент масиву вказує на початок суцільного списку блоків з описами назв, що відповідають цьому геш-значенню. `NULL` означає, що жодне ім'я не відповідає вказаному значенню.



Блок є списком всередині структури з покажчиками на назву, текст заміни та наступний блок у списку. Наступний нульовий покажчик позначає кінець списку.

```
struct nlist {              /* запис таблиці: */
      struct nlist *next;         /* наступний запис в ланцюжку */
      char *name;                 /* визначене ім'я */
      char *defn;                 /* текст заміни */
};
```

Масив покажчиків, це просто

```
#define HASHSIZE 101

static struct nlist *hashtab[HASHSIZE];   /* таблиця покажчиків */
```

Функція гешування, використовувана обома, `lookup` та `install`, додає кожне значення символу ланцюжка до зашифрованої комбінації попередніх, і повертає частку поділу на розмір гешу. Це не найкраща з можливих функцій гешування, але вона коротка й ефективна.

```
/* hash:  утворює геш-значення для ланцюжка s */
unsigned hash(char *s)
{
    unsigned hashval;

    for (hashval = 0; *s != '\0'; s++)
        hashval = *s + 31 * hashval;
    return hashval % HASHSIZE;
}
```

Беззнакова арифметика забезпечує додатнє геш-значення.

Процес гешування видасть початковий індекс для масиву `hashtab`; якщо ланцюжок існує, його можна буде знайти у списку блоків, які там беруть свій початок. Пошук здійснюється функцією `lookup`. Якщо `lookup` знайде відповідний запис, вона поверне покажчик на нього; якщо ні — `NULL`.

```
/* lookup:  шукає s у hashtab */
struct nlist *lookup(char *s)
{
    struct nlist *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np;          /* знайдено */
    return NULL;                /* не знайдено */
}
```

Цикл `for` функції `lookup` є стандартною ідіомою проходження вздовж зв'язного списку:

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
    ...
```

функція `install` використовує `lookup`, щоб визначити, чи назва, яка додається вже присутня; якщо так, то нове визначення витіснить старе. У протилежному випадку, буде створено новий запис. `install` поверне `NULL`, якщо з якоїсь причини не залишилося місця для нового запису.

```
struct nlist *lookup(char *);
char *strdup(char *);

/* install: додає (name, defn) до hashtab */
struct nlist *install(char *name, char *defn)
{
    struct nlist *np;
    unsigned hashval;

    if ((np = lookup(name)) == NULL) { /* not found */
        np = (struct nlist *) malloc(sizeof(*np));
            if (np == NULL || (np->name = strdup(name)) == NULL)
                return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    } else            /* already there */
        free((void *) np->defn);       /*free previous defn */
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}
```

Вправа 6-5. Напишіть функцію `undef`, яка би видаляла назву та визначення з таблиці, утворюваної функціями `lookup` та `install`.

Вправа 6-6. Втільте просту версію (без аргументів) оброблювача визначень `#define` (який можна би було використовувати із С-програмами), основуючись на функціях із цього розділу. Вам можливо доведеться згадати функції `getch` і `ungetch`.

6.7 Typedef

С передбачає засіб, що має назву `typedef`, який дозволяє створювати нові назви типів даних. Так, наприклад, оголошення

```
typedef int Length;
```

робить назву `Length` синонімом `int`. Тепер тип `Lengh` можна вживати в оголошеннях, зведеннях тощо, так само як ми це робимо з `int`:

```
Length len, maxlen;
Length *lengths[];
```

Подібно до цього, оголошення

```
typedef char *String;
```

перетворює `String` на синонім `char *` (символьний покажчик), що пізніше можна використати в оголошеннях і зведеннях типів:

```
String p, lineptr[MAXLINES], alloc(int);
int strcmp(String, String);
p = (String) malloc(100);
```

Зверніть увагу, що новий тип, який було оголошено в `typedef`, з'являється на місці назви змінної, а не одразу після слова `typedef`. Синтаксично, `typedef` подібний до зберігальних класів `extern`, `static` тощо. Ми використали заголовні перші літери нових типів, щоб вони вирізнялися.

Як складніший приклад, ми могли би застосувати `typedef` для деревовидних вузлів з попереднього розділу:

```
typedef struct tnode *Treeptr;

typedef struct tnode { /* the tree node: */

    char *word;               /* points to the text */
    int count;                /* number of occurrences */
    struct tnode *left;       /* left child */
    struct tnode *right;      /* right child */
} Treenode;
```

Це утворить дві нові назви типів із назвою `Treenode` (структура) і `Treeptr` (покажчик на структуру). Після цього, функція `talloc` виглядатиме як

```
Treeptr talloc(void)
{
    return (Treeptr) malloc(sizeof(Treenode));
}
```

Слід однак підкреслити, що оголошення `typedef` не створює, власно кажучи, цілком нового типу — воно просто додає нову назву для того, який вже існує. Також, семантика не зазнає жодних змін — змінні, оголошені в такий спосіб, матимуть ті самі властивості, що й змінні, чий тип вказано безпосередньо. Фактично, `typedef` аналогічний `#define` за винятком того, що, через те, що його інтерпретовано компілятором, він може впоратися з текстовими замінами, які значно перевищують можливості препроцесора. Наприклад,

```
typedef int (*PFI)(char *, char *);
```

утворює тип `PFI`, який є покажчиком на функцію (з двома аргументами `char *`), і повертає ціле. Це можна використати в такому контексті, як

```
PFI strcmp, numcmp;
```

скажімо, програми `sort` з Розділу 5.

Попри чисто естетичні міркування, існують дві вагоміші причини використання `typedef`. Перша, це параметризувати програму від проблем з портабельністю. При використанні `typedef` з типами даних, які можуть виявитись машинозалежними, потрібно тільки поміняти `typedef`, коли програму перенесено на іншу машину. Поширеним випадком є використання `typedef`-назв для різноманітних цілих величин, після чого встановлення відповідних значень для `short`, `int` та `long` для кожної окремої машини. Типи на зразок `size_t` та `ptrdiff_t` зі стандартної бібліотеки є хорошим прикладом.

Другою ціллю `typedef` є надання кращої документації програмі — тип із назвою `Treeptr` легше зрозуміти ніж такий, що оголошено просто як покажчик на складну структуру.

6.8 Сполуки

Сполука — це змінна, яка може втримувати (в різний час) об'єкти різного типу та розміру, компілятор обчислюючи розміри та умови вирівнювання. Сполуки забезпечують можливістю маніпулювання різного роду даними, збереженими в єдиному місці, без потреби включення в програму якоїсь машинозалежної інформації. Вони аналогічні варійованим записам в Pascal.

Для прикладу, який можна віднайти, скажімо, в керівникові символів компілятора, припустімо, що константа може бути `int`, `float` або покажчиком на символьний масив. Значення певної константи потрібно зберегти у змінній відповідного типу, однак найзручнішим для керівника символів буде, щоб це значення займало ту саму кількість пам'яті та зберігалося в тому самому місці, незалежно від типу. Саме для цієї мети існують сполуки — єдина змінна, яка може легітимно утримувати будь-який із вказаних типів. Синтаксис основується на тому, що притаманний структурам:

```
union u_tag {
    int ival;
    float fval;
    char *sval;
} u;
```

Змінна `u` буде достатньо великою, щоб утримати найбільше з цих трьох типів; певний розмір залежить від реалізації. Будь-який з цих типів можна присвоїти `u` і потім використати в якомусь виразі за умови, що використання буде несуперечливим: отриманий тип повинен збігатися з типом, збереженим останнього разу. Саме на програміста падає відповідальність за тим, щоб стежити за тим який тип в дану мить збережено в сполуці; результат залежить від реалізації, якщо щось збережено як один тип, а добуто як інший.

Синтаксично доступ до членів сполуки здійснюється як

```
назва-сполуки.член
```

або

```
покажчик-на-сполуку->член
```

точно так, як це відбувається в структурах. Якщо для слідкування за тим який тип у дану мить збережено в сполуці `u` використати змінну `utype`, тоді ви можете зустріти код подібний на наступне

```
if (utype == INT)
    printf("%d\n", u.ival);
if (utype == FLOAT)
    printf("%f\n", u.fval);
if (utype == STRING)
    printf("%s\n", u.sval);
else
    printf("bad type %d in utype\n", utype);
```

Сполуки можна зустріти всередині структур і масивів, а також навпаки. Синтаксис доступу до члена сполуки розміщеного в структурі (і навпаки) тотожний гніздованим структурам. Наприклад, в структурному масиві, означеному як

```
struct {
    char *name;
    int flags;
    int utype;
```

```
        union {
            int ival;
            float fval;
            char *sval;
        } u;
    } symtab[NSYM];
```

до члена `ival` можна звернутися як

```
symtab[i].u.ival
```

а до першого символу ланцюжка `sval` одним із наступних

```
*symtab[i].u.sval
symtab[i].u.sval[0]
```

Насправді сполука — це структура, в якій всі члени мають зміщення відносно бази рівне нулю, структура достатньо велика, щоб утримати найбільшого члена, і вирівнювання придатне для всіх типів у сполуці. Стосовно сполук так само як структур застосовні ті самі операції: присвоєння або копіювання як цілого, здобуття адреси і доступ до одного з членів. Сполуку можна ініціювати тільки величиною того самого типу що її перший член; таким чином сполуці u вище можна присвоїти початкове значення тільки у вигляді `int`.

Розподільник пам'яті з Розділу 8 демонструє як можна використати сполуку, щоб змусити вирівнювання змінної з певною межею пам'яті.

6.9 Розрядні поля

Коли з пам'яттю справді скрутно, то може з'явитися потреба помістити декілька об'єктів в єдине машинне слово; одним із поширених використань, наприклад, є набір однобітних прапорців у додатках на зразок таблиці символів компілятора. Формати даних нав'язані ззовні, як у випадку інтерфейсів до апаратного устаткування, також часто вимагають здатності одержати частини слова.

Уявіть фрагмент компілятора, що маніпулює таблицею символів. Кожний ідентифікатор у програмі має певну інформацію, пов'язану з ним, наприклад чи це ключове слово чи ні, чи зовнішнє чи/або статичне, і так далі. Найбільш компактним способом закодувати таку інформацію, це встановити однобітний прапорець на єдиному `char` або `int`.

Звичний спосіб за допомогою якого це можна зробити, це означити набір «маск», що відповідатимуть слушним позиціям бітів, як наприклад

```
#define KEYWORD       01
#define EXTRENAL      02
#define STATIC        04
```

або

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

Числа повинні бути в степені двійки. Після цього доступ до бітів складатиметься з їхнього «перебирання» шляхом зсуву, маскування, і доповнювальних операторів описаних в Розділі 2.

Певні ідіоми зустрічаються досить часто:

```
flags |= EXTERNAL | STATIC;
```

вмикає біти `EXTERNAL` і `STATIC` у `flags`, тоді як

```
flags &= ~(EXTERNAL | STATIC);
```

вимикає їх, і

```
if ((flags & (EXTERNAL | STATIC)) == 0) ...
```

є істинним, якщо обидва біти вимкнено.

Незважаючи на легкість засвоєння цих ідіом, як альтернатива C пропонує можливість безпосереднього означення та доступу полів всередині слова замість порозрядних логічних операторів. Розрядне поле, або просто «поле» для скорочення, є набором суміжних бітів всередині єдиної, визначеної реалізацією пам'ятової одиниці, яке ми називаємо «словом». Синтаксис означення полів і доступу до них основується на притаманному структурам. Наприклад, означення `#define` вище можна замінити означеннями трьох полів:

```
struct {
    unsigned int is_keyword : 1;
    unsigned int is_extern  : 1;
    unsigned int is_static  : 1;
} flags;
```

Це означує змінну під назвою `flags`, яка містить три 1-бітних поля. Число, яке слідує за двокрапкою відображає ширину поля в бітах. Поля оголошено як `unsigned int`, щоб упевнитися, що це беззнакові величини.

Звертання до окремих полів таке саме як до інших членів структур: `flags.is_keyword`, `flags.is_extern` тощо. Поля містять ніби маленькі цілі і можуть брати участь в арифметичних операціях так само як інші цілі числа. Таким чином, попередній приклад більш природньо написати як

```
flags.is_extern = flags.is_static = 1;
```

щоб увімкнути біти,

```
flags.is_extern = flags.is_static = 0;
```

щоб вимкнути їх, і

```
if (flags.is_extern == 0 && flags.is_static == 0)
    ...
```

щоб перевірити.

Майже все, що пов'язано з полями залежить від втілення. Чи може поле виходити за межі слова залежить від реалізації. Поля не обов'язково повинні мати назви; безіменні поля (тільки двокрапка та ширина) застосовуються як заповнювачі. Щоб добитися вирівнювання з наступною межею слова, можна використати спеціальну ширину поля 0.

Присвоєння значення полям відбувається зліва направо на одних машинах, і з права на ліво на інших. Це означає, що незважаючи на те, що поля корисні для утримування внутрішньо-означених структур даних, слід уважно розглянути питання який кінець йде першим, розбираючи зовнішньо-означені дані; програми, які залежать від таких речей не є портабельними. Поля можна оголосити тільки як `int`; для портабельності явно вкажіть `signed` (знакові) або `unsigned` (беззнакові). Вони не є масивами і не мають адрес, оператор `&` застосувати неможливо.

Розділ 7

Ввід і вивід

Ввід і вивід не є, власне, складовою частиною самої мови C, тож ми не досі наголошували на них у нашому викладі. Проте, програми взаємодіють зі своїм середовищем у набагато складніший спосіб ніж ті, що ми показали. В цьому розділі ми опишемо стандартну бібліотеку — набір функцій, які забезпечують можливістю вводу та виводу, оперування ланцюжками, керування пам'яттю, використання математичних функцій та розмаїття інших послуг для C-програм. Ми зосередимо нашу увагу на вводі і виводі.

Стандарт ANSI точно описує ці функції, тож вони можуть існувати в сумісній формі на будь-якій системі, де існує C. Програми, які обмежують свою взаємодію із системою до можливостей, наданих стандартною бібліотекою, можна переносити з однієї системи на іншу без якихось змін.

Властивості функцій бібліотеки зазначено в більш ніж двох десятках файлів заголовка; ми вже зустрічалися з деякими з них, включаючи <stdio.h>, <string.h> і <ctype.h>. Ми не зможемо представити цілу бібліотеку тут, оскільки ми більш зацікавлені в написанні програм, що можуть її використовувати. Саму бібліотеку детально описано в Додатку Б.

7.1 Стандартний ввід і вивід

Як ми зазначили в Розділі 1, бібліотека втілює просту модель текстового вводу і виводу. Текстовий потік складається з послідовності рядків, кожний рядок закінчується знаком нового рядка. Якщо система не працює в такий спосіб, бібліотека зробить все від неї залежне, щоб здавалося, що все саме так і є. Наприклад, бібліотека може перетворювати повернення каретки і переведення рядка на символ нового рядка при вводі і в зворотньому напрямку при виводі.

Найпростіший механізм вводу — це читати по одному символу за один раз зі стандартного вводу, як правило це клавіатура, за допомогою `getchar`:

```
int getchar(void)
```

`getchar` повертає наступний введений знак кожний раз як її викликано, або `EOF`, коли вона зіткнулася з кінцем файла. Символічну константу `EOF` визначено в <`stdio.h`>. Це значення, типово, дорівнює -1, але краще вживати `EOF`, щоб не залежати від певного значення. В багатьох середовищах, клавіатуру можна замінити на файл скориставшись з умовного знака <, який позначає перенаправлення вводу: якщо програма `prog` послуговується `getchar`, тоді команда

```
prog <infile
```

змушує `prog` читати символи з `infile`, натомість. Перемкнення вводу відбувається в такий спосіб, що сама програма `prog` знає про зміну; зокрема, ланцюжок «<`infile`» не включається як аргумент командного рядка в `argv`. Заміна вводу також залишається невидимою, якщо ввід надходить з іншої програми через конвеєр: на деяких системах, команда

```
otherprog | prog
```

запускає дві програми, `otherprog` і `prog`, і передає через конвеєр стандартний вивід `otherprog` стандартному вводу `prog`.

Функція

```
int putchar(int)
```

використовується для виводу: `putchar(c)` виводить символ c на стандартний вивід, типово, це екран. `putchar` повертає виведений символ, або `EOF`, якщо сталася помилка. Знову ж таки, вивід можна перенаправити у файл за допомогою >`filename`. Якщо програма використовує `putchar`, команда

```
prog >outfile
```

записує стандартний вивід до `outfile`, натомість. Якщо підтримуються конвеєри,

```
prog | anotherprog
```

поміщає стандартний вивід `prog` у стандартний ввід `anotherprog`. Вивід, спричинений `printf`, також знаходить свій шлях до стандартного пристрою виводу. Виклики `putchar` і `printf` можуть чергуватися — вивід відбувається в тій послідовності, в якій виклики відбуваються.

Кожний вихідний файл, що згадує якусь з функцій бібліотеки вводу/виводу, повинен містити рядок

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

перед тим як їх вживати. Коли назву файла заголовка включено в дужки < та >, пошук такого відбувається в стандартному наборі місць в системі (наприклад, на UNIX, типовим каталогом є `/usr/include`).

Багато програм читають тільки один потік вводу і записують тільки один виводу; для таких програм буде цілком достатньо здійснення вводу і виводу за допомогою `getchar`, `putchar` і `printf`, або, принаймні, цього вистачить для початку. Це особливо так, якщо перенаправлення використовується для під'єднання виводу однієї програми до вводу іншої. Наприклад, розглянемо програму `lower`, яка переводить свій ввід у нижній регістр:

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

main() /* lower: переводить ввід у нижній регістр */
{
    int c

    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

Функцію `tolower` визначено в <`ctype.h`>, вона перетворює літеру верхнього регістру у нижній і повертає решту символів незмінними. Як ми зазначили раніше, «функції» на зразок `getchar` і `putchar` з <`stdio.h`> і `tolower` з <`ctype.h`> часто бувають макросами, щоб запобігти витраті ресурсів на виклик функції для кожного символу. Ми покажемо як здійснити це в Розділі 8.5. Незалежно від того, як втілено функції <`ctype.h`> на окремих машинах, програми, що використовують їх, не вимушені знати деталей про набір символів.

Вправа 7-1. Напишіть програму, яка би перетворювала літери верхнього регістру на нижній або нижнього на верхній в залежності від того, за яким ім'ям її було викликано, на що вказуватиме `argv[0]`.

7.2 Форматований вивід - printf

Функція виводу, `printf`, обертає внутрішні значення на друковні знаки. Ми неформально застосовували `printf` у попередніх розділах. Опис, який ви знайдете тут, охоплює найтиповіші випадки використання функції, але не є повним; для всіх подробиць зверніться до Додатка Б.

```
int printf(char *format, arg1, arg2, ...);
```

`printf` перетворює, форматує і виводить свої аргументи на стандартний пристрій виводу згідно із форматом (`format`). Вона повертає кількість виведених символів.

Ланцюжок формату містить два типи об'єктів: звичайні символи, які буде скопійовано до вивідного потоку і вказівники перетворення, кожен з яких призводить до, власне, перетворення і виводу кожного наступного аргументу `printf`. Кожний вказівник перетворення починається з `%` і закінчується знаком перетворення. Між `%` і цим знаком можуть також знаходитись, в даній послідовності:

- Знак мінуса, що вказує вирівнювання з лівого боку перетвореного аргументу.
- Число, яке вказує мінімальну ширину поля. Перетворений аргумент буде виведено у полі, щонайменше, цієї ширини. Якщо треба, то простір буде заповнено ліворуч (або праворуч, якщо вказано вирівнювання зліва) для того, щоб добитися потрібної ширини поля.
- Крапка, яка відокремлює ширину поля від вказівника точності.
- Число точності, яке вказує максимальну кількість знаків, що буде виведено з ланцюжка, або кількість цифр після десяткової крапки у випадку числа з рухомою точкою, або мінімальну кількість цифр цілого числа.
- Знак `h`, якщо ціле буде виведене як коротке (`short`), або `l` (англійська літера «л»), якщо як довге (`long`).

Символи перетворення показано в Таблиці 7.1. Якщо знак після `%` не є вказівником перетворення, поводження залишиться невизначеним.

Ширина або точність можуть бути вказаними як `*`, у такому разі, значення обчислюється шляхом перетворення наступного аргументу (який повинен бути типу `int`). Наприклад, щоб вивести, щонайбільше, `max` знаків ланцюжка `s`, ми можемо написати

```
printf("%.*s", max, s);
```

Більшість перетворень формату було проілюстровано в попередніх розділах. Одним виключенням є точність, оскільки вона стосується ланцюжків. Наступна таблиця демонструє ефект різних означень для виводу «hello, world» (12 знаків). Ми додали двокрапки навколо кожного поля, щоб було видно їхній розмір.

```
:%s:                    :hello, world:
:%10s:                  :hello, world:
:%.10s:                 :hello, wor:
:%-10s:                 :hello, world:
:%.15s:                 :hello, world:
```

```
:%-15s:                 :hello, world   :
:%15.10s:               :     hello, wor:
:%-15.10s:              :hello, wor     :
```

Табл. 7.1: Основні перетворення `printf`

Знак	Тип аргументу; виводиться як
`d`, `i`	`int`; десяткове число.
`o`	`int`; беззнакове вісімкове число (без нуля попереду).
`x`, `X`	`int`; беззнакове шістнадцяткове число (без `0x` або `0X` попереду), використовуючи `abcdef` або `ABCDEF` замість `10`, ..., `15`.
`u`	`int`; беззнакове ціле число.
`c`	`int`; один символ.
`s`	`char *`; виводить знаки ланцюжка до `'\0'` або тієї кількості знаків, яку задано вказівником точності.
`f`	`double`; [-]*m.dddddd*, де кількість *d* задано вказівником точності (без задання — 6).
`e`, `E`	`double`; [-]*m.ddddddе+/-xx* або [-]*m.ddddddE+/-xx*, де кількість *d* задано вказівником точності (без задання — 6).
`g`, `G`	`double`; застосовувати `%e` або `%E`, якщо показник степеня менший за -4 або більший за або рівний точності; у протилежному випадку використовувати `%f`. Хвостові нулі та хвостова десяткова крапка не виводяться.
`p`	`void *`; покажчик (представлення залежить від реалізації).
`%`	жодного аргументу не перетворено, вивести `%`.

Застереження: `printf` використовує свій перший аргумент для того, щоб дізнатися як багато аргументів в цілому і який їхній тип. Вона заплутається і ви отримаєте неправильний результат, якщо недостатньо аргументів або вони неправильного типу. Вам також слід знати про відмінність цих двох викликів:

```
printf(s);          /* ЗАЗНАЄ НЕВДАЧІ, якщо s містить % */
printf("%s", s);   /* БЕЗПЕЧНИЙ для використання */
```

Функція `sprintf` здійснює ті самі перетворення, що й `printf`, але зберігає свій вивід у ланцюжку:

```
int sprintf(char *string, char *format, arg1, arg2, ...);
```

`sprintf` форматує аргументи `arg1`, `arg2` і так далі, відповідно до формату (`format`), як і раніше, але результат в ланцюжку (`string`) замість стандартного виводу; `string` повинен бути досить великим, щоб втримати результат.

Вправа 7-1. Напишіть програму, яка виводитиме довільний текст вводу у розумний спосіб. Як мінімум, вона повинна відображати неграфічні знаки як вісімкові або шістнадцяткові числа, в залежності від вашого бажання, окрім цього розбивати довгі рядки тексту.

7.3 Списки аргументів довільної довжини

Цей розділ містить втілення мінімальної версії `printf` для демонстрації того, як написати функцію, яка би обробляла список аргументів довільної довжини у машинонезалежний спосіб. Оскільки ми, головним чином, зацікавлені в опрацьовуванні аргументів, `minprintf` оброблятиме форматувальний ланцюжок і аргументи, але викликатиме дійсний `printf` для перетворень формату. Чинним оголошенням `printf` є

```
int printf(char *fmt, ...)
```

де `...` означає, що кількість і типи цих аргументів може відрізнятися. Оголошення `...` може тільки з'являтися в кінці списку аргументів. Нашу `minprintf` оголошено як

```
void minprintf(char *fmt, ...)
```

оскільки ми не повертатимемо відлік символів, як це робить `printf`.

Делікатний момент полягає в тому, як `minprintf` просувається через список аргументів, в той час як сам список навіть не має імені. Стандартний файл заголовка <`stdarg.h`> містить набір макросів, який визначає, як проходити список аргументів. Втілення цього файла заголовка може відрізнятися на різних машинах, але інтерфейс, який він представляє є однорідним.

Тип `va_list` використовується для оголошення змінної, яка посилатиметься почергі до кожного аргументу; в `minprintf` ця змінна називається `ap`, як скорочення від ¨argument pointer¨ (покажчик на аргумент). Макрос `va_start` ініціалізує ap таким чином, щоб вона вказувала на перший безіменний аргумент. Макрос потрібно викликати один раз до того, як використовувати `ap`. Необхідно, щоб був щонайменше один аргумент з іменем; останній названий аргумент використовується `va_start`, щоб розпочати роботу.

Кожний виклик `va_arg` повертає один аргумент і переводить `ap` до наступного; `va_arg` використовує назву типу, щоб визначити, який тип повернути і наскільки великим повинен бути крок. І, нарешті, `va_end` здійснює очистку, якщо треба. Її потрібно викликати до того, як програма поверне своє значення.

Ці властивості складають базу для нашої спрощеної `printf`:

```
#include <stdarg.h>
```

```
/* minprintf:    мінімальна printf зі списком аргументів *
 * змінної довжини                                        */
void minprintf(char *fmt, ...)
{
    va_list ap; /* покажчик на кожний безіменний аргумент *
                 * по черзі                               */
    char *p, *sval;
    int ival;
    double dval;

    va_start(ap, fmt); /* заставляє ap бути покажчиком *
                        * на 1-ий безіменний аргумент  */
    for (p = fmt; *p; p++) {
        if (*p != '%') {
            putchar(*p);
            continue;
        }
        switch (*++p) {
        case 'd':
            ival = va_arg(ap, int);
            printf("%d", ival);
            break;
        case 'f':
            dval = va_arg(ap, double);
            printf("%f", dval);
            break;
        case 's':
            for (sval = va_arg(ap, char *); *sval; sval++)
                putchar(*sval);
                break;
        default:
            putchar(*p);
            break;
        }
    }
    va_end(ap); /* очистка */
}
```

Вправа 7-3. Виправіть `minprintf` так, щоб вона втілювала додаткові можливості `printf`.

7.4 Форматований ввід - scanf

Функція `scanf` є аналогічною `printf`, але тільки для вводу. Вона надає багато з тих самих можливостей перетворення в зворотньому, правда, напрямку.

```
int scanf(char *format, ...)
```

`scanf` зчитує знаки зі стандартного вводу, інтерпретуючи їх відповідно до специфікації, вказаній форматом (`format`), і зберігає результат за допомогою решти аргументів. Аргумент формату описано нижче; решта аргументів, кожен з яких повинен бути покажчиком, вказує на те, де відповідний перетворений ввід потрібно зберегти. Так само як і з `printf`, цей розділ є підсумком найкорисніших рис, а не вичерпним описом.

`scanf` зупиняється, якщо вона вичерпає свій список формату, або коли ввід не збігається із контрольною специфікацією. Вона повертає як значення число елементів вводу, що зійшлися і яких було присвоєно. Це можна використати, щоб взнати, скільки об'єктів було знайдено. При завершенні файла повертається `EOF`; зауважте, що не те саме що й `0`, який означає, що наступний символ вводу не зійшовся із першим описом в ланцюжку форматування. Наступний виклик `scanf` відновить продовжить пошук, починаючи з місця, де було оброблено останній знак.

Існує також функція `sscanf`, яка читає свій ввід з ланцюжка замість стандартного вводу:

```
int sscanf(char *string, char *format, arg1, arg2, ...)
```

Вона сканує ланцюжок, відповідно до формату `format` і зберігає отримані значення в `arg1`, `arg2` і так далі. Останні мають бути покажчиками.

Ланцюжок формату, як правило, містить описи перетворення, що використовуються для керування перетворенням вводу. Ланцюжок формату може містити:

- Пробіли і табуляцію, які не ігноруються.
- Звичайні знаки (не `%`), які повинні зійтися з наступним символом, який не є пробілом, з потоку вводу.
- Описувачі перетворення, що складаються зі знака `%`, необов'язкового знака блокування присвоєння `*`, необов'язкового числа, яке вказує ширину поля, необов'язкових `h`, `l` або `L`, які вказують ширину адресата та символ перетворення.

Описувач перетворення описує перетворення наступного поля вводу. Звичайно, результат розміщено в змінній, на яку вказує відповідний аргумент. Якщо ж за допомогою `*` вказано блокування присвоєння, ввідне поле пропускається, присвоєння не відбувається. Ввідним полем вважається ланцюжок знаків, які не є пробілами; воно продовжиться або до наступного пробілу, або доки ширину поля, якщо вказано,

вичерпано. Це означає, що `scanf` читатиме крізь границі для того, щоб знайти ввід, оскільки символи нового рядка також вважаються пробілами. (Символами-пробілами вважаються пробіл, табуляція, новий рядок, повернення каретки, вертикальна табуляція і зміна сторінки.)

Символ перетворення визначає інтерпретацію ввідного поля. Відповідний аргумент повинен бути покажчиком, як вимагається семантикою «виклику за за значенням» С. Символи перетворень показано в Таблиці 7.2).

Табл. 7.2: Основні перетворення `scanf`

Символ	**Вводимі дані; тип аргументу**
`d`	десяткове ціле; `int *`.
`i`	ціле; `int *`. Ціле може бути у вісімковій (з `0` попереду) або шістнадцятковій (з `0x` або `0X`) формі.
`o`	вісімкове ціле (із або без переднього `0`); `int *`.
`u`	беззнакове десяткове ціле; `unsigned int *`.
`x`	шістнадцяткове ціле (із або без попереднього 0x або 0X); `int *`.
`c`	символи; `char *`. Наступні введені знаки (без задання 1) розміщено у вказане місце. Звичайний пробіл пригнічено; щоб прочитати наступний не-пробіл, використайте `%1s`.
`s`	символьний ланцюжок (не екранований); `char *`, вказуючи на масив символів достатньо великий для ланцюжка і кінцевого `'\0'`, який буде додано.
`e`, `f`, `g`	число з рухомою точкою з необов'язковим знаком, необов'язковою експонентою; `float *`.
`%`	буквальний %; присвоєння не відбувається

Перед знаками перетворення `d`, `i`, `o`, `u` та `x` може стояти `h`, щоб вказати, що в списку аргументів знаходиться покажчик на коротке ціле (`short`) а не ціле (`int`), або `l` (англійська «л»), щоб вказати на покажчик на довге ціле (`long`).

Як перший приклад, простенький калькулятор з Розділу 4 можна написати зі `scanf`, щоб здійснювати перетворення вводу:

```
#include <stdio.h>

main()          /* простий калькулятор */
{
    double sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
```

```
}
```

Скажімо, ми хочемо прочитати рядки вводу, які містять дату у формі

```
25 Dec 1988
```

Твердження зі `scanf` у такому разі становитиме

```
int day, year;
char monthname[20];

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

Знак `&` не використовується зі змінною, що зберігає назву місяця `monthname`, оскільки назва масиву і так є покажчиком. Буквальні знаки також можуть з'являтися в ланцюжку формату `scanf`; вони повинні зійтися з такими самими знаками у вводі. Тож ми могли би читати дати, що мають форму `мм/дд/рр` за допомогою виразу зі `scanf`:

```
int day, month, year;

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

`scanf` ігнорує пробіли і табуляцію в ланцюжкові формату. Більше того, вона пропускає пропуски і пробіли (пробіли, табуляцію, нові рядки тощо), розглядаючи ввід. Для прочитання вводу, чий формат не є сталим, кращим буде читати по одному рядкові за раз, після чого розбити його на окремі частини за допомогою `scanf`. Наприклад, скажімо ми хотіли би прочитати рядки, які можуть включати дату в одній з, наведених вище, форм. В такому разі ми могли би написати

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
        printf("valid: %s\n", line); /* форма 25 Dec 1988 */
    else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
        printf("valid: %s\n", line); /* форма мм/дд/рр */
    else
        printf("invalid: %s\n", line); /* недійсна форма */
}
```

Виклики `scanf` можна змішувати з викликами інших функцій вводу. Наступний виклик будь-якої ввідної функції почнеться з прочитання першого знака, не прочитаного `scanf`.

Останнє попередження: аргументи `scanf` і `sscanf` повинні бути покажчиками. Найчастішою помилкою є написання

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

замість

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

Цю помилку, як правило, не буде виявлено під час компіляції.

Вправа 7-4. Напишіть власну версію `scanf`, аналогічну `minprintf` з попереднього розділу.

Вправа 5-5. Перепишіть postfix-калькулятор з Розділу 4 так, щоб він використовував `scanf` і/або `sscanf` для вводу і перетворення чисел.

7.5 Доступ до файлів

Приклади, з якими ми досі стикалися, усі читали стандартний ввід і записували до стандартного виводу, які автоматично визначаються для програми операційною системою.

Наступним кроком буде написати програму, яка матиме доступ до файла. Одна з програм, що демонструє необхідність таких операцій, це `cat`, яка зчеплює набір вказаних їй файлів, виводячи їх на стандартний пристрій виводу. `cat` використовується для видруку файлів на екрані і як збирач вводу загального призначення для програм, які не мають можливості доступу до файлів за іменем. Наприклад, команда

```
cat x.c y.c
```

виводить вміст файлів `x.c` та `y.c` (і нічого більше) на стандартний пристрій виводу. Питання в тому, як забезпечити прочитання даних файлів, тобто, як під'єднати зовнішні назви, про які думає користувач, до твердження, яке читатиме дані.

Правила — прості. Перед тим як його можна прочитати або здійснити до нього запис, файл потрібно відкрити за допомогою бібліотечної функції `fopen`. `fopen` візьме зовнішні назви, такі як `x.c` або `y.c`, здійснить певні службові дії і переговори з операційною системою (деталі яких не повинні нас хвилювати), і повертає покажчик, який використовуватиметься в наступних читаннях і записах до файла.

Цей покажчик, який називається покажчиком файла, вказує на структуру, яка містить інформацію про файл, таку як місцеперебування буфера, поточне положення знака в буфері, чи файл читається, чи до нього йде запис, і, чи мали місце помилки або вказівник кінця файла. Користувачі не повинні знати подробиць, оскільки визначення, що знаходяться в <`stdio.h`> включають оголошення структури під назвою `FILE`. Єдине, що потрібно знати, це оголошення покажчика файла, яке спрощено виглядає як

```
FILE *fp;
FILE *fopen(char *name, char *mode);
```

Це означає, що `fp` є покажчиком на (структуру типу) `FILE` і `fopen` повертає покажчик на `FILE`. Замітьте, що `FILE`, це назва типу, така сама як `int`, а не мітка структури; її означено за допомогою `typedef`. (Деталі щодо того як можна втілити fopen на UNIX надано в Розділі 8.5).) Виклик `fopen` у програмі має вигляд

```
fp = fopen(name, mode);
```

Першим аргументом `fopen` є символьний ланцюжок, що містить назву файла. Другий аргумент — це режим, також символьний ланцюжок, який вказує на те, як буде використовуватись файл. Допустимими режимами можуть бути режим читання («`r`» — read), запису («`w`» — write) і долучення («`a`» — append). Деякі системи розрізняють текстові і бінарні файли, для останніх потрібно додати «`b`» (binary) до літери режиму.

Якщо файл, якого не існує, відкрито для запису або додання, його буде створено, якщо це можливо. Відкриття наявного файла для запису спричинить звільнення старого змісту, тоді як відкриття для дозапису (доточення) — збереже старий зміст. Спроба прочитання файла, якого не існує, викличе помилку; існують також інші причини помилок, як скажімо спроба прочитати файл, на який ви не маєте дозволу. Якщо сталася якась помилка, `fopen` повертає `NULL`. (Характер помилки можна визначити точніше; подивіться обговорення функцій з обробки помилок у Розділі 1 Додатка Б.)

Наступний крок — мати спосіб читання або запису до файла після того, як його відкрито. `getc` повертає наступний знак з файла; вона вимагає покажчика на файл, щоб знати, який саме файл.

```
int getc(FILE *fp)
```

`getc` повертає наступний знак з потоку, на який вказує `fp`; вона повертає `EOF` у випадку кінця файла або помилки.

`putc` — це функція виводу:

```
int putc(int c, FILE *fp)
```

`putc` запише символ з `c` до файла `fp` і поверне записаний знак або `EOF` у разі помилки. Так само як `getchar` і `putchar`, `getc` із `putc` можуть бути макросами, а не функціями. Під час запуску С-програми, середовище операційної системи бере на себе завдання відкриття трьох файлів і надання покажчиків до них. Ці файли — це стандартний ввід, стандартний вивід і стандартна помилка; відповідні їм покажчики

називаються stdin, stdout і stderr, і оголошено в <stdio.h>. За звичайних обставин, stdin сполучено з клавіатурою, тоді як stdout із stderr — з екраном, але stdin і stdout можна перенаправити в інші файли або конвеєри, як описано в Розділі 7.1.

getchar і putchar можна визначити через getc, putc, stdin і stdout наступним чином:

```
#define getchar()           getc(stdin)
#define putchar(c)          putc((c), stdout)
```

Для форматованого вводу або виводу із файлами, можна використати функції fscanf і fprintf. Вони тотожні scanf із printf за винятком того, що першим аргументом є покажчик на файл, який вказує на, власне, файл, який читатиметься; другим аргументом є ланцюжок формату.

```
int fscanf(FILE *fp, char *format, ...)
int fprintf(FILE *fp, char *format, ...)
```

Маючи такий начерк, ми тепер можемо написати власну програму cat для зчеплення файлів. Схема буде тією самою, що виявилась корисною в багатьох інших програмах. Якщо існують аргументи командного рядка, їх буде інтерпретовано як назви файлів і оброблено по-порядку. Якщо немає аргументів, обробляється стандартний ввід.

```
#include <stdio.h>

/* cat:    зчеплює файли, 1-а версія */
main(int argc, char *argv[])
{
      FILE *fp;
      void filecopy(FILE *, FILE *)

      if (argc == 1) /* немає аргументів; копіює *
                      * стандартний ввід          */
          filecopy(stdin, stdout);
      else
         while(--argc > 0)
               if ((fp = fopen(*++argv, "r")) == NULL) {
                    printf("cat: can't open %s\n, *argv);
                    return 1;
                } else {
                    filecopy(fp, stdout);
                    fclose(fp);
```

```
                }
        return 0;
}

/* filecopy:    копіює файл ifp до файла ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
    int c;

    while ((c = getc(ifp)) != EOF)
            putc(c, ofp);
}
```

Покажчики на файл `stdin` і `stdout` є об'єктами типу `FILE *`. Незважаючи на це, вони є константами (сталими), а не змінними, що внеможливлює надання їм нового значення.

Функція

```
int fclose(FILE *fp)
```

— це протилежність `fopen`, вона розриває зв'язок між покажчиком на файл і зовнішньою назвою, отриманою `fopen`, звільняючи покажчик для іншого файла. Оскільки більшість операційних систем мають певне обмеження кількості файлів, які програма спроможна відкрити одночасно, хорошою ідеєю буде звільнити покажчики на файли, якщо їх більше не потрібно, як ми це зробили в `cat`. Існує також інша причина зробити `fclose` для файла виводу — це очищує буфер, в якому `putc` накопичує вивід. `fclose` викликається автоматично для кожного відкритого файла при нормальному завершенні програми. (Ви можете замкнути `stdin` і `stdout`, якщо вони непотрібні. Їх також можна переназначити за допомогою бібліотечної функції `freopen`.)

7.6 Обробка помилок - stderr і exit

Обробка помилок у `cat` не є ідеальною. Проблема в тому, що якщо неможливо дістатися до одного з файлів з якоїсь причини, діагностичне повідомлення буде виведене в кінці зчепленого виводу. Це, можливо, допустимо, якщо вивід направляється на екран, але не тоді, коли він надходить у файл або іншу програму через конвеєр.

Щоб краще справлятися з подібними ситуаціями, програмам окрім `stdin` і `stdout` надається ще один потік виводу під назвою `stderr`. Вивід, записаний до `stderr`, як правило, з'являється на екрані навіть якщо стандартний вивід перенаправлено.

Спробуймо виправити `cat` так, щоб писати її повідомлення про помилки до стандартної помилки.

```
#include <stdio.h>

/* cat:    зчеплює файли, 2-а версія */
main(int argc, char *argv[])
{
     FILE *fp;
     void filecopy(FILE *, FILE *);
     char *prog = argv[0]; /* назва програми, для помилок */

     if (argc == 1 ) /* немає аргументів; копіює  *
                      * стандартний ввід           */
          filecopy(stdin, stdout);
     else
          while (--argc > 0)
               if ((fp = fopen(*++argv, "r")) == NULL) {
                         fprintf(stderr, "%s: can't open %s\n",
                                 prog, *argv);
                         exit(1);
               } else {
                         filecopy(fp, stdout);
                         fclose(fp);
               }
     if (ferror(stdout)) {
          fprintf(stderr, "%s: error writing stdout\n", prog);
          exit(2);
     }
     exit(0);
}
```

Програма сигналізує про помилки в два способи. Перший: діагностичний вивід, спричинений `fprintf` надходить до `stderr`, тож він знаходить свій шлях на екран замість того, щоб зникнути кудись через конвеєр або в файлі виводу. Ми включили в повідомлення назву програми з `argv[0]`, тож, якщо програма використовується разом з іншими, джерело помилки буде ідентифіковано.

Другий: програма використовує функцію стандартної бібліотеки `exit`, яка завершує виконання програми, якщо її викликано. Аргумент `exit` стане доступний будь-якому процесові, який викликав даний, тож успіх чи невдача програми може перевірятись іншою програмою, яка використовує першу як дочірній процес. Традиційно, повернене значення 0 сигналізує, що все успішно; ненульові значення, звично, означають анормальні ситуації. `exit` викликає `fclose` для кожного відкритого файла виводу для того, щоб очистити будь-який буферований вивід.

Всередині `main`, `return` *вираз* еквівалентне `exit(`*вираз*`)`. Перевага використання

exit полягає в тому, що її можна викликати з інших функцій і її виклики можна знаходити за допомогою програм пошуку за шаблоном як ті, які ви знайдете у Розділі 5.

Функція ferror повертає ненульове значення, якщо сталася помилка при обробці потоку fp.

```
int ferror(FILE *fp)
```

Хоча помилки виводу — рідкість, вони теж стаються (наприклад, якщо диск заповнено до кінця), тому виробнича програма повинна це перевірити.

Функція feof(FILE *) є аналогічною ferror; вона повертає ненульове значення, якщо досягнуто кінця файла.

```
int feof(FILE *fp)
```

Загалом, нас не цікавив статус виходу наших маленьких ілюстративних програм, але будь-яка серйозна програма повинна піклуватися про повернення зрозумілих і корисних значень статусу.

7.7 Ввід і вивід рядків

Стандартна бібліотека передбачає функцію вводу і виводу fgets, аналогічну getline, яку ми використовували в попередніх розділах:

```
char *fgets(char *line, int maxline, FILE *fp)
```

fgets читає наступний рядок вводу (включаючи символ нового рядка) з файла fp у символьний масив line; щонайбільше maxline-1 знаків буде прочитано. Отриманий в результаті рядок буде завершено '\0'. Звично, fgets повертає рядок; у випадку кінця файла або помилки, вона повертає NULL. (Наша getline повертає довжину рядка, — корисніше значення; нуль означає кінець файла.)

Для виводу, функція fputs записує ланцюжок (який не повинен включати символ нового рядка) до файла:

```
int fputs(char *line, FILE *fp)
```

Вона повертає EOF, якщо сталася помилка, і додатнє значення у протилежному випадку.

Функції бібліотеки gets і puts тотожні fgets і fputs, але оперують над stdin і stdout. Плутанину викликає те, що gets видаляє кінцевий '\n', тоді як puts додає його.

Щоб продемонструвати, що немає нічого особливого в таких функціях як fgets і fputs, ось вони, будь ласка, скопійовані зі стандартної бібліотеки до нашої системи:

```
/* fgets:    отримує щонайбільше n символів з iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
          if ((*cs++ = c) == '\n')
               break;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}

/* fputs:    додає ланцюжок s до файла iop */
int fputs(char *s, FILE *iop)
{
    int c;

    while (c = *s++)
          putc(c, iop);
    return ferror(iop) ? EOF : 0;
}
```

З невідомих причин, стандарт вказує на відмінні значення повернення для `ferror` і `fputs`. Досить легко втілити нашу `getline` за допомогою `fgets`:

```
/* getline:    читає рядок, повертає довжину */
int getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
          return 0;
    else
          return strlen(line);
}
```

Вправа 7-6. Напишіть програму по порівнянню двох файлів, виводячи перший рядок, там, де вони відрізняються.

Вправа 7-7. Змініть програму знаходження по шаблону з Розділу 5 таким чином, щоб вона брала свій ввід з набору вказаних файлів, або, якщо жодного файла не вказано як аргумент, зі стандартного вводу. Можливо, можна також виводити назву файла, в якому знайдено рядок, який збігся.

Вправа 7-8. Напишіть програму видруку набору файлів, починаючи кожний на новій сторінці, з титулом і відліком сторінок для кожного файла.

7.8 Додаткові функції

Стандартна бібліотека надає широкий вибір різноманітних функцій. Цей розділ, це стислий перегляд найкорисніших з них. Додаткові деталі і багато інших функцій можна знайти в Додатку Б.

7.8.1 Операції з ланцюжками

Ми вже згадали ланцюжкові функції `strlen`, `strcpy`, `strcat` і `strcmp`, які ви знайдете в <`string.h`>. В наступному огляді `s` із `t` — це `char *`, тоді як `c` та `n` — це `int`.

`strcat(s,t)`	доточує `t` до кінця `s`
`strncat(s,t,n)`	доточує `n` знаків `t` до кінця `s`
`strcmp(s,t)`	повертає від'ємне, нуль або додатнє значення для `s < t`, `s == t` або `s > t`
`strncmp(s,t,n)`	те саме, що й `strcmp`, але тільки для перших `n` знаків
`strcpy(s,t)`	копіює `t` до `s`
`strncpy(s,t,n)`	копіює щонайбільше `n` символів `t` до `s`
`strlen(s)`	повертає довжину `s`
`strchr(s,c)`	повертає покажчик на перший знайдений символ `c` у `s`, або `NULL`, якщо жодного не знайдено
`strrchr(s,c)`	повертає покажчик на останній знайдений символ `c` у `s`, або `NULL`, якщо жодного не знайдено

7.8.2 Перевірка та перетворення класів символів

Декілька функцій з <`ctype.h`> здійснюють перевірку символів і перетворення. В наступному, `c` є типу `int`, який може бути представленим як `unsigned char` (беззнаковим символом) або `EOF`. Функції повертають `int`.

`isalpha(c)`	повертає ненульове значення, якщо `c` є літерою алфавіту, і 0 — якщо ні
`isupper(c)`	повертає ненульове значення, якщо `c` є літерою верхнього регістру, і 0 — якщо ні
`islower(c)`	повертає ненульове значення, якщо `c` є літерою нижнього регістру, і 0 — якщо ні
`isdigit(c)`	повертає ненульове значення, якщо `c` є цифрою, і 0 — якщо ні
`isalnum(c)`	повертає ненульове значення, якщо `isalpha(c)` або `isdigit(c)` істинні, і 0 — якщо ні
`isspace(c)`	повертає ненульове значення, якщо `c` є пробілом, табуляцією, символом нового рядка, вертанням каретки, зміною сторінки або вертикальною табуляцією
`toupper(c)`	повертає літеру `c`, обернену у верхній регістр
`tolower(c)`	повертає літеру `c`, обернену у нижній регістр

7.8.3 Ungetc

Стандартна бібліотека надає досить обмежену версію функції `ungetch`, яку ми написали у Розділі 4; вона називається `ungetc`.

```
int ungetc(int c, FILE *fp)
```

Вона проштовхує символ `c` назад у файл `fp` і повертає або `c`, або `EOF` у випадку помилки. Тільки один символ на файл гарантовано проштовхнути назад. `ungetc` може використовуватись разом з кожною з ввідних функцій, таких як `scanf`, `getc` або `getchar`.

7.8.4 Виконання команд

Функція `system(char *s)` виконує команду, що міститься в символьному ланцюжку `s`, після чого продовжує виконання поточної програми. Вміст `s` залежить великою мірою від операційної системи. Як тривіальний приклад, на UNIX-системах, вираз

```
system("date");
```

спричиняє до запуску програми `date`; вона видруковує дату і час дня на стандартному виводі. `system` повертає системозалежне ціле статусу виконаної команди. На UNIX, статус буде значенням, поверненим `exit`.

7.8.5 Керування пам'яттю

Функції `malloc` і `calloc` динамічно добувають відрізки пам'яті.

```
void *malloc(size_t n)
```

повертає покажчик на `n` байтів неініційованої пам'яті, або `NULL`, якщо запит неможливо задовольнити.

```
void *calloc(size_t n, size_t size)
```

повертає покажчик на досить вільного місця для масиву з `n` об'єктами вказаного розміру `size`, або `NULL`, якщо запит неможливо задовольнити. Місце зберігання ініціюється нулем.

Покажчик, повернений `malloc` або `calloc`, матиме відповідне вирівнювання для даного об'єкту, але його потрібно звести до належного типу, як скажімо

```
int *ip;

ip = (int *) calloc(n, sizeof(int));
```

`free(p)` звільняє місце, на яке вказує `p`, де `p` попередньо отримано викликом `malloc` або `calloc`. Немає обмежень щодо послідовності в якій звільнюється простір, але виникне жахлива помилка, якщо звільнити щось, що не було отримано викликом `malloc` або `calloc`.

Помилкою буде також використати щось після того, як воно було звільнено. Типовий код з помилкою, в цьому циклові, що звільняє елементи зі списку:

```
for (p = head; p != NULL; p = p->next) /* НЕПРАВИЛЬНО */
    free(p);
```

Правильним буде зберегти те, що потрібно перед тим як звільнювати:

```
for (p = head; p != NULL; p = q) {
    q = p->next;
    free(p);
}
```

У Розділі 8.7 ви знайдете втілення розподільника пам'яті, як от `malloc`, в якому відведені блоки може бути звільнено в будь-якій послідовності.

7.8.6 Математичні функції

Існує більш ніж двадцять математичних функцій, оголошених в <`math.h`>; ось декілька з найчастіше використовуваних. Кожна з них візьме один або два аргументи типу `double` (подвійного) і повертає теж `double`.

`sin(x)`	синус x, x в радіанах
`cos(x)`	косинус x, x в радіанах
`atan2(y,x)`	арктангенс y/x, в радіанах
`exp(x)`	показникова функція e^x
`log(x)`	натуральний логарифм x *($x>0$)* (при основі e)
`log10(x)`	десятковий (звичайний) логарифм x *($x>0$)*
`pow(x,y)`	x^y
`sqrt(x)`	квадратний корінь x *($x>0$)*
`fabs(x)`	абсолютне значення x

7.8.7 Генератор випадкових чисел

Функція `rand()` обчислює послідовність псевдовипадкових цілих в діапазоні від нуля до `RAND_MAX`, означеного в <`stdlib.h`>. Одним з способів здобуття випадкових чисел з рухомою точкою більших або рівних нулю але менших одиниці, це

```
#define frand() ((double) rand() / (RAND_MAX+1.0))
```

(Якщо ваша бібліотека передбачає функцію для випадкових чисел з рухомою точкою, вона ймовірніше матиме кращі статистичні властивості ніж наведена вище.)

Функція `srand(unsigned)` встановлює зерно для `rand`. Портабельне втілення `rand` і `srand`, рекомендоване Стандартом, можна знайти в Розділі 2.7.

Вправа 7-9. Функції на зразок `isupper` можна втілити так, щоб зберегти місце або зберегти час. Розгляньте обидві можливості.

Розділ 8

Інтерфейс системи UNIX

Операційна система UNIX надає свої сервіси через посередництво набору системних викликів, котрі насправді є функціями, вбудованими в операційну систему, які можуть бути викликані користувацькими програмами. В цьому розділі знаходиться опис того, як використати деякі з найважливіших системних викликів з С-програм. Якщо ви користуєтесь UNIX, це повинно бути безпосередньою допомогою, оскільки часом буває потрібно застосувати системний виклик для максимальної ефективності, або мати доступ до певної властивості, яку ви не знайдете в бібліотеці. Навіть якщо ви використовуєте С на відмінній операційній системі, то маєте можливість заглянути в нутро С-програмування через вивчення цих прикладів; хоча деталі можуть відрізнятися, подібний код можна знайти на будь-якій системі. Оскільки бібліотеку ANSI С, в багатьох випадках, модельовано за властивостями UNIX, цей код допоможе вам також зрозуміти саму бібліотеку.

Даний розділ розділено на три основні частини: ввід/вивід, файлова система і відведення пам'яті. Перші дві частини передбачають поверхове знайомство із зовнішніми характеристиками UNIX-систем.

У Розділі 7 описано, однорідний для різноманітних операційних систем, інтерфейс вводу/виводу. На певній операційній системі, функції стандартної бібліотеки будуються з урахуванням можливостей, наданих підставовою системою. В наступних декількох розділах ми розглянемо системі виклики UNIX для вводу та виводу, і продемонструємо як за їхньої допомоги втілити деякі частини стандартної бібліотеки.

8.1 Дескриптори файлів

В операційній системі UNIX, весь ввід і вивід здійснюється за допомогою читання та запису до файлів, оскільки всі периферійні пристрої, навіть клавіатура й екран — це файли, які є частиною основної файлової системи. Це означає, що єдиний однорідний інтерфейс забезпечує комунікацію між програмами та зовнішніми пристроями.

Загалом, перед тим як читати або здійснити запис до файла, вам потрібно повідомити операційну систему про свої наміри — процес, що називається *відкриттям*

файла. Якщо ви збираєтесь здійснити запис до файла, то може виникнути потреба спочатку створити його або очистити попередній зміст. Система перевірить ваше право на такі дії (Чи файл існує? Чи маєте ви дозвіл на доступ до нього?), і якщо все гаразд, поверне програмі невелике додатнє ціле — *дескриптор файла*. У випадку вводу чи виводу до файла, замість назви використовуватиметься дескриптор файла для його ідентифікації. (Дескриптор файла є аналогічним покажчику на файл, застосовному в стандартній бібліотеці, або індексу файла з MS-DOS.) Вся інформація про відкритий файл зберігатиметься операційною системою, а користувацька програма посилатиметься на файл тільки через дескриптор.

Оскільки ввід і вивід, включаючи клавіатуру й екран, настільки поширені, існує певна організація, щоб зробити їх зручнішими. Коли інтерпретатор команд («оболонка») запускає програму, три файли з дескрипторами 0, 1 і 2 відкрито під назвою стандартний ввід, стандартний вивід і стандартна помилка. Якщо програма читає 0 і записує до 1 і 2, вона спроможна здійснювати ввід і вивід, не турбуючись про відкриття додаткових файлів.

Користувач програми може перенаправляти ввід/вивід до та з файлів за допомогою < і >:

```
prog <infile >outfile
```

В цьому випадку, оболонка поміняє стандартні призначення дескрипторів файлів 0 і 1 на вказаний файл (`outfile`). За звичайних обставин, дескриптор 2 залишається приєднаним до екрана, тож повідомлення про помилки можуть надходити туди. Подібні спостереження стосуються вводу та виводу, переданих через конвеєр. В усіх випадках, призначення файлів міняється оболонкою, а не програмою. Програма ж навіть не знатиме ні звідки надходить її ввід, ні куди йде вивід доти, поки вона користується файлом 0 для вводу і 1 і 2 — для виводу.

8.2 Низькорівневий ввід/вивід - read і write

Операції вводу та виводу застосовують системні виклики `read` і `write`, доступ до яких з С-програм здійснюється за допомогою двох функцій під назвою `read` і `write`. Для обох, першим аргументом є дескриптор файла. Другий аргумент — це символьний масив, вказаний вашою програмою, куди надійдуть дані або звідки їх можна отримати. Третій аргумент — це кількість байтів, яку буде передано.

```
int n_read = read(int fd, char *buf, int n);
int n_written = write(int fd, char *buf, int n);
```

Кожний виклик поверне відлік числа переданих байтів. При читанні, кількість повернених байтів може виявитися меншою за вказану. Повернення нуль байтів означає кінець файла, а `-1` вказує на якусь помилку. При запису, значення, що повертається,

дорівнюватиме кількості записаних байтів; якщо значення не дорівнює кількості запитаних байтів, це означатиме помилку.

Будь-яка кількість байтів може бути прочитана або записана за один раз. Найпоширенішим значенням є 1, що означає по одному символу за раз («небуферований ввід або вивід»), а також 1024 або 4096, що відповідає фізичному розмірові блока зовнішнього пристрою. Більші розміри ефективніші, позаяк це зменшує кількість системних викликів.

Маючи цю інформацію, ми можемо написати просту програму копіювання власного вводу до виводу, еквівалентну програмі копіювання файлів з Розділу 1. Ця програма копіюватиме будь-що до будь-чого, оскільки ввід і вивід можна перенаправити до будь-якого файла або пристрою.

```
#include "syscalls.h"

main() /* копіює ввід до виводу */
{
    char buf[BUFSIZ];
    int n;

    while ((n = read(0, buf, BUFSIZ)) > 0)
        write(1, buf, n);
    return 0;
}
```

Ми об'єднали прототипи функцій системних викликів у один файл під назвою `syscalls.h` для включення його в програми в цьому розділі. Проте, ця назва не є стандартною. Параметр `BUFSIZ` також визначено в `syscalls.h`; його значення — це розмір, який підходить для нашої системи. Якщо розмір файла не є кратним `BUFSIZ`, деякі виклики `read` можуть повернути меншу кількість байтів для запису `write`; наступний виклик `read` поверне нуль, в такому випадку.

Корисно побачити, як можна використати `read` і `write` для створення функцій вищого рівня, таких як `getchar`, `putchar` тощо. Наприклад, ось версія `getchar`, яка здійснює небуферований ввід шляхом читання стандартного вводу по одному символові за раз.

```
#include "syscalls.h"

/* getchar: небуферований ввід по одному знакові */
int getchar(void)
{
    char c;

    return (read(0, &c, 1) == 1) ? (unsigned char) c : EOF;
```

```
}
```

змінна `c` повинна мати тип `char`, оскільки `read` вимагає знакового покажчику. Зведення `c` до `unsigned char` (беззнакового символу) усуває будь-які проблеми зі знаками.

Наступна версія `getchar` читає ввід великими відрізками, а виводить символи поодному за раз.

```
#include "syscalls.h"

/* getchar: проста буферована версія */
int getchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int n = 0;

    if (n == 0) { /* буфер порожній */
        n = read(0, buf, sizeof buf);
        bufp = buf;
    }
    return (--n >= 0) ? (unsigned char) *bufp++ : EOF;
}
```

Якщо ці версії `getchar` було би скомпільовано зі включеним <`stdio.h`>, необхідно би було `#undef` (скасувати) назву `getchar` у випадку, якщо її втілено як макрос.

8.3 Open, creat, close, unlink

Попри типові стандартний ввід, вивід і помилку, ви повинні явно відкрити файли для того, щоб читати або записувати до них. Існують два системні виклики для цього: `open` і `creat` (саме так, а не «create»). `open` чимось схожий на функцію `fopen`, розглянуту в Розділі 7, за винятком того, що замість повертати покажчик на файл, вона повертає дескриптор, — просто ціле. `open` видасть `-1`, якщо сталася якась помилка.

```
#include <fcntl.h>

int fd;
int open(char *name, int flags, int perms);

fd = open(name, flags, perms);
```

Так само як і у випадку з `fopen`, аргумент `name` — це символьний ланцюжок, що містить назву файла. Другий аргумент, `flags`, є цілим (`int`), що вказує режим відкриття файла; основними значеннями є

`O_RDONLY`	відкрити тільки для читання
`O_WRONLY`	відкрити тільки для запису
`O_RDWR`	відкрити як для читання, так і для запису

Ці константи на System V UNIX визначено в <`fcntl.h`>, і в <`sys/file.h`> на версіях Берклі (BSD).

Щоб відкрити файл, який існує, для читання:

```
fd = open(name, O_RDONLY,0);
```

Аргумент дозволу (`perms`) завжди дорівнюватиме нулеві для користувачів `open`, ми це обговоримо пізніше. Помилкою буде намагатися відкрити файл, якого не існує. Для створення нових файлів або перезапису старих є системний виклик `creat`

```
int creat(char *name, int perms);

fd = creat(name, perms);
```

який вертає дескриптор файла, якщо виклик спромігся створити його, і `-1` — якщо ні. Якщо файл вже існує, `creat` зітне його довжину до нуля, звільняючи його попередній зміст таким чином, тож це не вважається помилкою — створити файл, який вже існує.

Якщо файла до цього не існувало, `creat` створить його з дозволами, вказаними аргументом `perms`. У файловій системі UNIX, існує дев'ять бітів інформації про дозволи, пов'язаних з файлами, які керують читанням, записом і можливістю виконання власником файла, групою, до якої він належить, і рештою користувачів. Таким чином, зручно вказати дозволи як трьохзначне вісімкове число. Наприклад, `0775` вказує на дозвіл на читання, запис і виконання власником, читання і виконання для групи і решти користувачів.

Для ілюстрації, розглянемо спрощену версію UNIX-програми `cp`, яка копіює один файл до іншого. Наша версія копіює тільки один файл, вона не дозволяє, щоб другим аргументом був каталог, і вона вигадує власні дозволи замість зберігати їх.

```
#include <stdio.h>
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666    /* читання й запис для власника, *
                       * групи й решти              */

void error(char *,      ...);
```

```
/* cp:    копіює f1 до f2 */
main(int argc, char *argv[])
{
    int f1, f2, n;
    char buf[BUFSIZ];

    if (argc != 3)
        error("Usage: cp from to");
    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
          error("cp: can't create %s, mode %03o",
         argv[2], PERMS);
    while ((n = read(f1, buf, BUFSIZ)) > 0)
         if (write(f2, buf, n) != n)
              error("cp: write error on file %s", argv[2]);
    return 0;
}
```

Ця програма створює файл зі сталими правами доступу `0666`. Завдяки системному викликові `stat`, розглянутому в Розділі 8.6, ми можемо визначити дозволи файла, який існує, і таким чином забезпечити тими самими копіями.

Зверніть увагу, що функцію `error` викликано зі змінним списком аргументів, дуже схоже на `printf`. Втілення `error` ілюструє, як використати ще одного члена сімейства `printf`. Функція стандартної бібліотеки `vprintf` схожа на `printf`, за винятком того, що змінний список аргументів замінено на єдиний аргумент, який ініціюється викликом макросу `va_start`. Подібним чином, `vfprintf` і `vsprintf` відповідають `fprintf` із `sprintf`.

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

/* error:    виводить повідомлення про помилку і *
 * завершує роботу                                */
void error(char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "error: ");
    vprintf(stderr, fmt, args);
```

```
    fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

Існує обмеження (часто, коло 20-и) кількості файлів, які програма може одночасно відкрити. Відповідно, будь-яка програма, яка має намір обробити багато файлів, повинна бути готовою до багаторазового використання того самого дескриптора. Функція `close(int fd)` розриває зв'язок між дескриптором і відкритим файлом, і звільняє дескриптор для використання іншим файлом; вона відповідає `fclose` зі стандартної бібліотеки, за винятком браку буфера для очищення. Завершення програми через `exit` або `return` у `main` закриє всі відкриті файли.

Функція `unlink(char *name)` видаляє назву файла з файлової системи. Вона відповідає функції стандартної бібліотеки `remove`.

Вправа 8-1. Перепишіть програму `cat` з Розділу 7, використовуючи `read`, `write`, `open` і `close` замість їхніх еквівалентів зі стандартної бібліотеки. Поекспериментуйте щодо перевірки відносної швидкості двох версій програми.

8.4 Довільний доступ - lseek

Звично, ввід і вивід є послідовними: кожне читання або запис має місце в положенні всередині файла одразу за попереднім. Проте якщо потрібно, файл можна прочитати або записати в довільній послідовності. Системний виклик `lseek` дозволяє переміщатися всередині файла, навіть не читаючи чи записуючи жодних даних:

```
long lseek(int fd, long offset, int origin);
```

що встановлює поточну позицію файла, чий дескриптор дорівнює `fd`, до `offset` (зміщення), яке обчислюється відносно положення, вказаного `origin`. Послідовне читання чи запис почнеться саме з цієї позиції. `origin` може дорівнювати 0, 1 або 2, щоб вказати, що зміщення (`offset`) обчислюватиметься з початку, з поточної позиції, або з кінця файла, відповідно. Наприклад, для додання (доточування) до файла (перенаправлення >> в оболонці UNIX, або «`a`» у випадку `fopen`), знайдіть кінець до того як записувати:

```
lseek(fd, 0L, 2);
```

Щоб повернутися на початок («перемотати»):

```
lseek(fd, 0L, 0);
```

Зверніть увагу на аргумент `0L`; його можна було би записати як `(long) 0`, або просто як `0`, якщо `lseek` оголошено відповідним чином.

Завдяки `lseek`, файли можна розглядати приблизно як масиви, ціною будучи повільніший доступ. Наприклад, наступна функція читає будь-яку кількість байтів з будь-якого місця всередині файла. Вона повертає прочитане число, або `-1` при помилці.

```
#include "syscalls.h"

/* get:    читає n байтів, починаючи з позиції pos */
int get(int fd, long pos, char *buf, int n)
{
     if (lseek(fd, pos, 0) >= 0) /* переміщає в pos */
          return read(fd, buf, n);
     else
          return -1;
}
```

Значення, повернене `lseek`, буде число типу `long`, яке вказує на нове положення всередині файла, або `-1`, якщо сталася помилка. Функція стандартної бібліотеки `fseek` є аналогічною `lseek` за винятком того, що першим аргументом є `FILE *`, а повернене значення є просто ненульовим при помилці.

8.5 Приклад: втілення fopen і getc

Давайте проілюструємо як ці частини скласти разом шляхом втілення функцій стандартної бібліотеки `fopen` і `getc`.

Якщо пригадуєте, файли в стандартній бібліотеці описано за покажчиками на файл, а не дескрипторами. Покажчик на файл — це просто покажчик на структуру, яка містить декілька деталей інформації про файл: покажчик на буфер, тож файл можна читати великими відрізками; відлік кількості знаків, що залишились у буфері; покажчик на наступну позицію знака в буфері; дескриптор файла; і, нарешті, прапорці, що описують режим (читання/запису), статус помилки тощо.

Структура даних, що описує файл, міститься в <`stdio.h`>, який треба включити (за допомогою `#include`) в будь-який файл, що використовує функції зі стандартної бібліотеки вводу/виводу. Вона також включена в функції цієї бібліотеки. В наступному вирізку з типового <`stdio.h`>, назви, призначені для використання тільки функціями бібліотеки, починаються з жорсткого пробілу, тож зменшується ймовірність того, що вони зійдуться із назвами в користувацьких програмах. Ця умовність застосовується в усіх функціях стандартної бібліотеки.

```
#define NULL                0
```

```
#define EOF                 (-1)
#define BUFSIZ              1024
#define OPEN_MAX            20  /* max #files open at once */

typedef struct _iobuf {
      int       cnt;            /* characters left */
      char *ptr;                /* next character position */
      char *base;               /* location of buffer */
      int       flag;           /* mode of file access */
      int       fd;             /* file descriptor */
} FILE;
extern FILE _iob[OPEN_MAX];

#define stdin             (&_iob[0])
#define stdout            (&_iob[1])
#define stderr            (&_iob[2])

enum _flags {
      _READ       = 01,         /* file open for reading */
      _WRITE      = 02,         /* file open for writing */
      _UNBUF      = 04,         /* file is unbuffered */
      _EOF        = 010,        /* EOF has occurred on this file */
      _ERR        = 020         /* error occurred on this file */
};

int _fillbuf(FILE *);
int _flushbuf(int, FILE *);

#define feof(p)                  ((p)->flag & _EOF) != 0)
#define ferror(p)                ((p)->flag & _ERR) != 0)
#define fileno(p)                ((p)->fd)

#define getc(p)            (--(p)->cnt >= 0 \
                      ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
#define putc(x,p) (--(p)->cnt >= 0 \
                      ? *(p)->ptr++ = (x) : _flushbuf((x),p))

#define getchar()                getc(stdin)
#define putcher(x)               putc((x), stdout)
```

Макрос `getc`, як правило, зменшує відлік, просуває покажчик і повертає знак. (Якщо пам'ятаєте, довший рядок `#define` можна перенести за допомогою зворотньої

похилої.) Проте, якщо відлік виявиться від'ємним, `getc` викликає функцію `_fillbuf` для дозаповнення буфера, ініціювання наново змісту структури і повернення знака. Символи повертаються як беззнакові, щоб упевнитись, що усі вони додатні.

Хоч ми не обговорюватимемо деталей, ми включили також визначення putc, щоб показати, що вона діє подібно до `getc`, викликаючи функцію `_flushbuf`, коли її буфер заповнено. Ми включили так само макрос визначення статусу помилки і кінця файла і дескриптор.

Тепер ми можемо написати функцію `fopen`. В основному, `fopen` піклується про відкриття файла і правильне розміщення, а також встановлення бітів прапорців для вказівки чинного режиму. `fopen` не відводить місця під буфер; це здійснюється `_fillbuf` під час першого прочитання файла.

```
#include <fcntl.h>
#include "syscalls.h"
#define PERMS 0666   /* RW for owner, group, others */

FILE *fopen(char *name, char *mode)
{
     int fd;
     FILE *fp;

     if (*mode != 'r' && *mode != 'w' && *mode != 'a')
          return NULL;
     for (fp = _iob; fp < _iob + OPEN_MAX; fp++)
          if ((fp->flag & (_READ | _WRITE)) == 0)
                 break;           /* found free slot */
     if (fp >= _iob + OPEN_MAX)     /* no free slots */
          return NULL;

     if (*mode == 'w')
          fd = creat(name, PERMS);
     else if (*mode == 'a') {
          if ((fd = open(name, O_WRONLY, 0)) == -1)
                 fd = creat(name, PERMS);
          lseek(fd, 0L, 2);
     } else
          fd = open(name, O_RDONLY, 0);
     if (fd == -1)           /* couldn't access name */
          return NULL;
     fp->fd = fd;
     fp->cnt = 0;
     fp->base = NULL;
     fp->flag = (*mode == 'r') ? _READ : _WRITE;
```

```
        return fp;
    }
```

Ця версія `fopen` не оперує всіма режимами доступу, зазначених стандартом, але додання їх не займе багато коду. Зокрема, наша `fopen` не розпізнає «`b`», що позначає бінарний доступ, оскільки це не має змісту на системах UNIX, так само «`+`», що дозволяє одночасно читання і запис. Перший виклик `getc` для певного файла зіткнеться з відліком рівним нулю, що спричинить виклик `_fillbuf`. Якщо `_fillbuf` виявить, що файл недоступний для читання, вона негайно поверне `EOF`. У протилежному випадку, вона спробує виділити буфер (якщо читання буфероване).

Як тільки започатковано буфер, `_fillbuf` викличе `read`, щоб заповнити його, встановить лічильник і покажчики, і поверне перший символ буфера. Наступні виклики `_fillbuf` матимуть виділений буфер.

```
    #include "syscalls.h"

    /* _fillbuf:       allocate and fill input buffer */
    int _fillbuf(FILE *fp)
    {
        int bufsize;

        if ((fp->flag&(_READ|_EOF_ERR)) != _READ)
            return EOF;
        bufsize = (fp->flag & _UNBUF) ? 1 : BUFSIZ;
        if (fp->base == NULL)               /* no buffer yet */
            if ((fp->base = (char *) malloc(bufsize)) == NULL)
                return EOF;                 /* can't get buffer */
        fp->ptr = fp->base;
        fp->cnt = read(fp->fd, fp->ptr, bufsize);
        if (--fp->cnt < 0) {
            if (fp->cnt == -1)
                fp->flag |= _EOF;
            else
                fp->flag |= _ERR;
            fp->cnt = 0;
            return EOF;
        }

        return (unsigned char) *fp->ptr++;
    }
```

Єдиним не з'ясованим питанням є — як все розпочати. Потрібно спершу означити і ініціювати масив `_iob` для `stdin`, `stdout` і `stderr`:

```
FILE _iob[OPEN_MAX] = {                 /* stdin, stdout, stderr */
      { 0, (char *) 0, (char *) 0, _READ, 0 },
      { 0, (char *) 0, (char *) 0, _WRITE, 1 },
      { 0, (char *) 0, (char *) 0, _WRITE, | _UNBUF, 2 }
};
```

Ініціалізація прапорців структури показує нам, що `stdin` читатиметься, `stdout` записуватиметься і `stderr` записуватиметься небуферовано.

Вправа 8-2. Перепишіть `fopen` і `_fillbuf` з полями замість явних операцій з бітами. Порівняйте розмір коду і швидкість виконання.

Вправа 8-3. Розробіть і напишіть `_flushbuf`, `fflush` і `fclose`.

Вправа 8-4. Функція стандартної бібліотеки

```
int fseek(FILE *fp, long offset, int origin)
```

є ідентичною `lseek` за винятком того, що `fp` — це покажчик на файл замість дескриптора і повернене значення є статусом типу `int`, а не позицією. Напишіть `fseek`. Впевніться, що ваша `fseek` відповідно узгоджується з буферуванням, що має місце в решті функцій бібліотеки.

8.6 Приклад - перелік вмісту каталогів

Іноді буває потрібний відмінний тип взаємодії з файловою системою — здобуття інформації про самий файл, а не про його зміст. Команда переліку змісту каталогу, така як ls Unix, є прикладом того — вона виводить назви файлів каталогу, і, за бажанням, іншу інформацію, таку як розміри, дозволи тощо. Аналогічною є команда `dir` системи MS-DOS. Оскільки каталог в Unix, це також файл, `ls` повинна тільки прочитати його, щоб отримати назви файлів, що там знаходяться. Зате, щоб здобути іншу інформацію, як скажімо розмір, виникає потреба в системному викликові. На деяких операційних системах, навіть для того, щоб отримати назви файлів, необхідно звернутися до системного виклику; у випадку того самого MS-DOS, наприклад. Що нам потрібно, так це надати доступ до такої інформації у відносно системонезалежний спосіб, навіть якщо реалізація буде дуже системозалежною.

Ми проілюструємо дещо з цього шляхом написання програми під назвою `fsize`. `fsize`, буде спеціальною формою `ls`, яка виводитиме розміри всіх файлів, вказаних їй на командному рядкові. Якщо один з аргументів виявиться каталогом, `fsize` викличе себе рекурсивно стосовно цього каталогу. Коли не вказано жодних аргументів, вона оброблятиме поточний каталог.

Давайте розпочнемо з короткого огляду файлової системи Unix. Каталог — це файл, що містить список назв файлів і певну інформацію щодо їхнього розташування. «Розташування», це індекс з іншої таблиці під назвою «список індексних вузлів». Індексний вузол файла містить всю інформацію про файл за виключенням його назви. Записи в каталогах, звичайно, складаються з двох пунктів — назви файла та номера індексного вузла.

На жаль, формат і вміст каталогу не однаковий на різних версіях системи. Тож ми розіб'ємо завдання на дві частини, і намагатимемося ізолювати непортабельні частини. Зовнішній рівень означить структуру під назвою `Dirent`, і три функції: `opendir`, `readdir` і `closedir`, щоб забезпечити системонезалежним доступом до назви й індексного вузла файла. Ми складемо `fsize` саме із таким інтерфейсом. Після цього, ми покажемо як втілити таку функцію на системах, що використовують таку саму структуру каталогів, як Version 7 і System V UNIX; інші варіанти залишено як вправа читачеві.

Структура `Dirent` включає номер індексного вузла та назву файла. Максимальна довжина назви файла дорівнюватиме `NAME_MAX`, що є системозалежним значенням. `opendir` повертає покажчик на структуру із назвою `DIR`, аналогічну `FILE`, використовувану `readdir` і `closedir`. Ця інформація зберігатиметься у файлі `dirent.h`.

```
#define NAME_MAX        14  /* longest filename component; */
                                        /* system-dependent */
typedef struct {                /* portable directory entry */
    long ino;                               /* inode number */
    char name[NAME_MAX+1];      /* name + '\0' terminator */
} Dirent;

typedef struct {        /* minimal DIR: no buffering, etc. */
    int fd;           /* file descriptor for the directory */
    Dirent d;                       /* the directory entry */
} DIR;

DIR *opendir(char *dirname);
Dirent *readdir(DIR *dfd);
void closedir(DIR *dfd);
```

Системний виклик `stat` візьме назву файла та поверне всю інформацію індексного вузла, пов'язаного з цим файлом, або `-1`, якщо мала місце помилка. Тобто

```
char *name;
struct stat stbuf;
int stat(char *, struct stat *);

stat(name, &stbuf);
```

заповнить структуру `stbuf` інформацією індексного вузла для вказаної назви файла. Структура з описом значень, повернених `stat` знаходиться в <`sys/stat.h`>, і типово виглядає як наступне:

```
struct stat        /* inode information returned by stat */
{
    dev_t       st_dev;             /* device of inode */
    ino_t       st_ino;             /* inode number */
    short       st_mode;            /* mode bits */
    short       st_nlink;           /* number of links to file */
    short       st_uid;             /* owners user id */
    short       st_gid;             /* owners group id */
    dev_t       st_rdev;            /* for special files */
    off_t       st_size;            /* file size in characters */
    time_t      st_atime;           /* time last accessed */
    time_t      st_mtime;           /* time last modified */
    time_t      st_ctime;           /* time originally created */
};
```

Більшість цих значень пояснено в полі коментарів. Такі типи як `dev_t` та `ino_t` визначено в <`sys/types.h`>, який також потрібно включити.

Пункт `st_mode` містить набір прапорців з описом файла. Визначення цих прапорців так само знаходиться в <`sys/types.h`>; нас цікавить тільки частина, що відповідає типові файла:

```
#define S_IFMT          0160000         /* type of file: */
#define S_IFDIR         0040000         /* directory */
#define S_IFCHR         0020000         /* character special */
#define S_IFBLK         0060000         /* block special */
#define S_IFREG         0010000         /* regular */
/* ... */
```

Тепер ми готові до написання програми `fsize`. Якщо режим, отриманий за допомогою `stat`, вказує на те, що файл не є каталогом, тоді розмір знаходиться тут-таки, і можна вивести безпосередньо. Однак, якщо назва вказує на каталог, тоді нам доведеться обробити цей каталог, один файл за раз; він може, в свою чергу, містити інші каталоги, тож цей процес буде рекурсивним.

Функція `main` матиме справу з аргументами командного рядка; вона передасть кожний аргумент функції `fsize`.

```
#include <stdio.h>
#include <string.h>
#include "syscalls.h"
#include <fcntl.h>          /* прапорці читання та запису */
#include <sys/types.h>      /* визначення типів */
#include <sys/stat.h>       /* структура, що повертається stat */
#include "dirent.h"

void fsize(char *)

/* виводить назву файла */
main(int argc, char **argv)
{
     if (argc == 1)      /* стандартно: поточний каталог */
          fsize(".");
     else
          while (--argc > 0)
               fsize(*++argv);
     return 0;
}
```

Функція `fsize` виводить розмір файла. Якщо ж файл є каталогом, `fsize` спершу викличе `dirwalk` для обробки всіх файлів, що там знаходяться. Зверніть увагу на те, як використовуються прапорці `S_IFMT` і `S_IFDIR` для визначення того, чи є файл каталогом. Дужки важливі, оскільки пріоритет `&` є нижчим за `==`.

```
int stat(char *, struct stat *);
void dirwalk(char *, void (*fcn) (char *));

/* fsize:     виводить назву файла "name" */
void fsize(char *name)
{
    struct stat stbuf;

    if (stat(name, &stbuf) == -1) {
        fprintf(stderr, "fsize: can't access %s\n", name);
        return;
    }
    if ((stbuf.st_mode & S_IFMT) == S_IFDIR)
        dirwalk(name, fsize);
    printf("%8ld %s\n", stbuf.st_size, name);
}
```

Функція `dirwalk`, це загальна рутина, яка викликає функцію для кожного файла всередині каталогу. Вона відкриває каталог, циклічно проходить через кожний файл всередині, викликаючи функцію для кожного з них, після чого закриває каталог та завершує роботу. Оскільки `fsize` звертається до `dirwalk` для кожного каталогу, ці дві функції викликають одна одну рекурсивно.

```
#define MAX_PATH 1024

/* dirwalk:  застосує fcn для всіх файлів у dir */
void dirwalk(char *dir, void (*fcn)(char *))
{
     char name[MAX_PATH];
     Dirent *dp;
     DIR *dfd;

     if ((dfd = opendir(dir)) == NULL) {
          fprintf(stderr, "dirwalk: can't open %s\n", dir);
          return;
     }
     while ((dp = readdir(dfd)) != NULL) {
          if (strcmp(dp->name, ".") == 0
               || strcmp(dp->name, ".."))
               continue; /* пропускаємо поточний і батьківський */
          if (strlen(dir)+strlen(dp->name)+2 > sizeof(name))
                fprintf(stderr, "dirwalk: name %s %s too long\n",
                      dir, dp->name);
          else {
                sprintf(name, "%s/%s", dir, dp->name);
                (*fcn)(name);
          }
     }
     closedir(dfd);
}
```

Кожний виклик `readdir` повертає покажчик на інформацію щодо наступного файла, або `NULL`, якщо не залишилося файлів. Кожний каталог завжди включає посилання на самого себе «`.`», і свій батьківський «`..`»; їх необхідно пропустити, інакше програма зациклиться.

До цього рівня код не залежить від того як сформовано каталоги. Наступним кроком буде представити мінімальні версії `opendir`, `readdir` і `closedir` для певної операційної системи. Наступні функції працюють з Version 7 та System V Unix-системами; вони використовують інформацію про каталоги з файла заголовка <`sys/dir.h`>, який має наступний вигляд:

```
#ifndef DIRSIZ
#define DIRSIZ 14
#endif
struct direct { /* запис каталогу */
    ino_t d_ino; /* номер індексного вузла */
    char d_name[DIRSIZ]; /* довга назва без '\0' */
};
```

Деякі версії цих систем дозволяють набагато довші назви і складнішу структуру каталогів.

Тип `ino_t` утворений за допомогою `typedef`, і вказує індекс зі списку вузлів. На нашій системі це значення відповідає `unsigned short`, але це не є тією інформацією яку варто на стале включати у вашу програму; воно може відрізнятися на іншій системі, тож typedef краще. Повний набір типів систем ви знайдете в <`sys/types.h`>.

`opendir` відкриває каталог, перевіряє чи файл дійсно є каталогом (цього разу за допомогою системного виклику `fstat`, подібного до `stat`, за винятком того, що він засовується до файлових дескрипторів), виділяє пам'ять під структуру каталогу, і занотовує інформацію:

```
int fstat(int fd, struct stat *);

/* opendir:     відкриває каталог для виклику readdir */
DIR *opendir(char *dirname)
{
    int fd;
    struct stat stbuf;
    DIR *dp;

    if ((fd = open(dirname, O_RDONLY, 0)) == -1
     || fstat(fd, &stbuf) == -1
     || (stbuf.st_mode & S_IFMT) != S_IFDIR
     || (dp = (DIR *) malloc(sizeof(DIR))) == NULL)
        return NULL;
    dp->fd = fd;
    return dp;
}
```

`closedir` закриває файл каталогу і звільняє місце:

```
/* closedir:     закриває каталог відкритий opendir */
void closedir(DIR * dp)
{
```

```
    if (dp) {
        close(dp->fd);
        free(dp);
    }
}
```

Нарешті, `readdir` користується `read`, щоб прочитати кожний запис в каталозі. Якщо каталоговий сегмент не використовується у даний момент (тому, що файл видалено), номер індексного вузла дорівнюватиме нулю, і ця позиція пропускається. У протилежному випадку, номер індексного вузла і назву буде занесено в статичну структуру, і покажчик на ці дані повернуто користувачеві. Кожний новий виклик перезаписує інформацію попереднього.

```
#include <sys/dir.h>            /* локальна структура директорій */

/* readdir:     читає записи директорій один за одним */
Dirent *readdir(DIR * dp)
{
    struct direct dirbuf;       /* локальна структура директорій */
    static Dirent d;            /* портабельна структура */

    while (read(dp->fd, (char *) &dirbuf, sizeof(dirbuf))
           == sizeof(dirbuf)) {
        if (dirbuf.d_ino == 0)  /* сегмент не використовується */
            continue;
        d.ino = dirbuf.d_ino;
        strncpy(d.name, dirbuf.d_name, DIRSIZ);
        d.name[DIRSIZ] = '\0';  /* забезпечує закінчення */
        return &d;
    }
    return NULL;
}
```

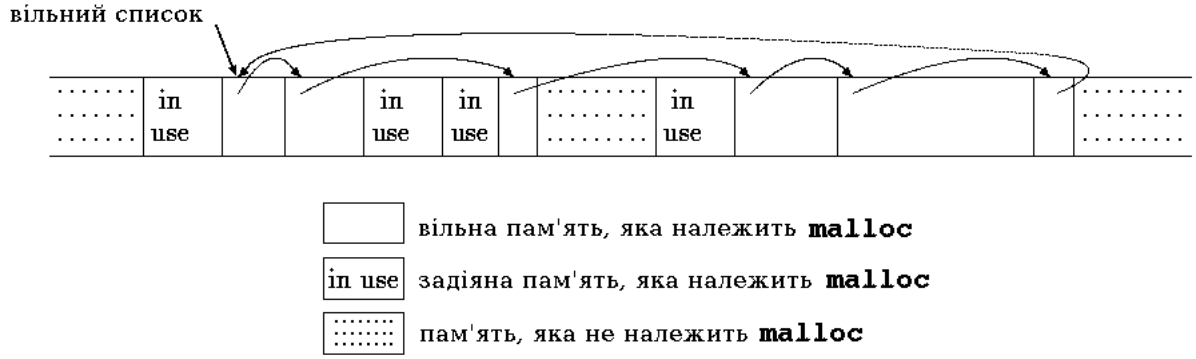
Незважаючи на те, що програма `fsize` досить спеціалізована, вона ілюструє декілька важливих ідей. Спершу, що багато програм не є «системними програмами»; вони просто користуються інформацією, утримуваної операційною системою. Для таких програм вирішальним є, щоб представлення інформації знаходилося тільки в стандартних заголовках, і щоб програми включали ці файли замість містити власні оголошення. Друге спостереження, що якщо постаратися, то можна створити інтерфейс до системозалежних об'єктів, який сам по собі буде відносно системонезалежним. Хорошим прикладом служать функції стандартної бібліотеки.

Вправа 8-5. Модифікуйте `fsize`, щоб вона виводила додаткову інформацію, що міститься у записі індексного вузла.

8.7 Приклад - розподільник пам'яті

У Розділі 5 ми представили дуже обмежений, орієнтований на стек розподільник пам'яті. Версія, яку ми зараз напишемо не має обмежень. Виклики `malloc` і `free` можуть відбуватися в будь-якій послідовності; `malloc` звертається до операційної системи, щоб отримати більше пам'яті, коли в цьому виникає необхідність. Ці функції демонструють деякі міркування при написанні машинонезалежного коду у відносно машинонезалежний спосіб, а також ілюструють реальне застосування структур, сполук і `typedef`.

Замість того, щоб призначити вкомпільований, сталого розміру масив, `malloc` звертатиметься із запитом щодо пам'яті до операційної системи по мірі необхідності. Оскільки інші дії в програмі можуть також вимагати місця без виклику розподільника, пам'ять яку відводить `malloc` може виявитись несуміжною. Тому вільне місце зберігається як список вільних блоків. Кожний блок включає інформацію про власний розмір, покажчик на наступний блок і, власне, вільне місце. Блоки зберігаються в послідовності зростання адрес пам'яті, і покажчик останнього блока (з найбільшою адресою) вказує на перший.



Під час запиту перевіряється вільний список доти, доки не буде знайдено досить великого блока. Цей алгоритм називається «алгоритмом першого збігу», на відміну від «алгоритму найкращого збігу», який шукає найменший блок, що задовольнить запит. Якщо розмір блока збігається з запитаним, його буде вилучено зі списку і повернено користувачеві. Якщо блок завеликий, його буде поділено і належна кількість повернено користувачеві, тоді як залишок зостанеться у вільному списку. Якщо не знайдено досить великого блока, операційна система надасть великий відрізок пам'яті, який буде приєднано до вільного списку.

Звільнення також спричиняє до пошуку списку вільної пам'яті, щоб знайти відповідне місце, щоб вставити звільнений блок. Якщо звільнений блок суміжний з вільним блоком з будь-якого боку, ці два буде об'єднано в один більший блок, щоб запобігти фрагментації пам'яті. Визначити суміжність легко, оскільки вільний список зберігається в послідовності зростання адрес.

Одна проблема, якої ми не торкнулися в Розділі 5, це впевненість, що пам'ять повернена `malloc` належно вирівняно для об'єктів, які там зберігатимуться. Незважаючи на те, що різні машини відрізняються, для кожної існує найбільший обмежувальний тип: якщо цей тип можна зберегти за певною адресою, то решту типів можна також. На деяких машинах найбільшим обмежувальним типом є `double`, на інших достатньо `int` або `long`.

Вільний блок містить покажчик на наступний вільний блок у низці, запис розміру блока і потім, власне, вільне місце; керівна інформація напочатку називається «заголовком». Для спрощення вирівнювання, всі блоки є кратними розміру заголовка, і самий заголовок вирівняно відповідно. Цього досягнуто за допомогою сполуки, яка містить відповідну структуру заголовка і приклад найбільшого обмежувального типу вирівнювання, який ми довільно зробили `long`:

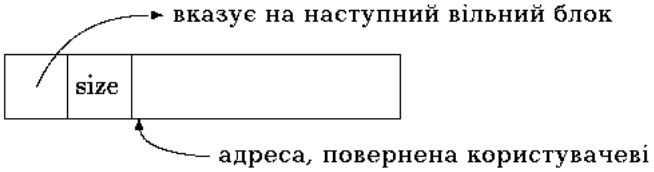
```
typedef long Align;      /* для вирівнювання до межі long */

union header {           /* заголовок блока */
   struct {
      union header *ptr; /* наступний блок, якщо є у списку вільних */
      unsigned size;     /* розмір цього блока */
   } s;
   Align x;              /* змусити вирівняти блоки */
};

typedef union header Header;
```

Поле `Align` ніколи не використовується, воно просто змушує, щоб кожний заголовок було вирівняно у межах найгіршого випадку.

У випадку `malloc`, запитаний розмір у символах округлюється вверх до відповідного числа одиниць заголовкового розміру; блок, який буде відведено міститиме одну або більше одиницю для самого заголовка, і це значення буде внесено в поле `size` заголовка. Покажчик, повернений `malloc` вказує на вільне місце пам'яті, а не на заголовок. Користувач може робити все, що йому заманеться із наданим вільним місцем, але, якщо щось записати поза межі призначеної пам'яті, найімовірніше, що список буде зіпсовано.



Блок, який повертає `malloc`

Поле `size` обов'язкове, оскільки блоки, контрольовані `malloc`, не обов'язково повинні бути неперервними, тож немає ніякої можливості обчислити розмір за допомогою покажчикової арифметики.

Для початку використовується змінна `base`. Якщо `freep` має значення `NULL`, як і повинно бути при першому виклику `malloc`, тоді буде утворено деградований список вільної пам'яті, що міститиме один блок розміром нуль і вказуватиме самий на себе. В будь-якому разі, після цього відбудеться пошук по списку вільної пам'яті. Пошук вільного блока відповідного розміру розпочнеться з місця (`freep`), де було знайдено останній блок; така стратегія допомагає зберегти список однорідним. Якщо знайдено завеликий блок, користувачеві буде повернено кінцеву (хвостову) частину, таким чином заголовок оригінальної повинен лиш змінити свій розмір. В усякому випадку, користувачу повертається покажчик, що вказує на вільне місце всередині блока, що бере свій початок з наступною одиницею після заголовка.

```
static Header base;             /* очищаємо список для початку */
static Header *freep = NULL;    /* започатковуємо список вільної
                                   пам'яті */

/* malloc:  розподільник пам'яті загального призначення */
void *malloc(unsigned nbytes)
{
    Header *p, *prevp;
    Header *moreroce(unsigned);
    unsigned nunits;

    nunits = (nbytes + sizeof(Header) - 1) / sizeof(header) + 1;
    if ((prevp = freep) == NULL) {  /* ще немає вільного списку */
        base.s.ptr = freeptr = prevptr = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr;; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {      /* достатньо великий */
            if (p->s.size == nunits)    /* точно */
                prevp->s.ptr = p->s.ptr;
            else {              /* виділити хвостову частину */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *) (p + 1);
        }
        if (p == freep)    /* звернений на себе вільний список */
```

```
            if ((p = morecore(nunits)) == NULL)
                return NULL;    /* нічого не залишилось */
        }
    }
```

Функція `morecore` добуває пам'ять від операційної системи. Подробиці щодо того як саме вона це здійснює можуть відрізнятися на різних системах. Це тому, що запит операційної системи щодо пам'яті, це доволі громіздка операція, ми не хочемо цього робити з кожним викликом `malloc`, тож `morecore` вимагає щонайменше `NALLOC` одиниць; цей більший блок буде розбито по мірі необхідності. Після встановлення поля `size`, `morecore` привносить додаткову пам'ять в поле зору, викликаючи `free`.

Системний виклик Unix `sbrk(n)` повертає покажчик на на `n` байтів більшу кількість пам'яті. `sbrk` повертає `-1`, якщо не залишилося місця, навіть якщо `NULL` було би кращим значенням. `-1` потрібно звести в `char *`, щоб його можна було порівняти з поверненим значенням. Знову ж таки, зведення роблять функцію відносно невразливою до деталей представлення покажчиків на різноманітних машинах. Проте існує одне припущення, що покажчики на різні блоки, повернені `sbrk`, можна змістовно порівнювати. Це не гарантовано стандартом, який який дозволяє порівняння покажчиків тільки в межах того самого масиву. Таким чином, ця версія `malloc` портабельна тільки серед машин, де загальне порівнювання покажчиків має сенс.

```
#define NALLOC    1024      /* мінімальна кількість одиниць, яку
                               слід запитати */

/* morecore:   звертається до системи по додаткову пам'ять */
static Header *morecore(unsigned nu)
{
    char *cp, *sbrk(int);
    Header *up;

    if (nu < NALLOC)
        nu = NALLOC;
    cp = sbrk(nu * sizeof(Header));
    if (cp == (char *) -1)      /* зовсім не залишилось місця */
        return NULL;
    up = (Header *) cp;
    up->s.size = nu;
    free((void *) (up + 1));
    return freep;
}
```

`free` сама по собі йде останньою. Вона сканує список вільної пам'яті, починаючи з `freep`, шукаючи місце, де можна вставити вільний блок. Це або між двома вже наявними блоками, або в кінці списку. В будь-якому разі, якщо звільнений блок прилягає до сусіднього, ці два блоки об'єднано в один. Єдина проблема в тому, щоб покажчики вказували на правильне місце і розміри були дійсними.

```
/* free:    додає блок ap до списку вільної пам'яті */
void free(void *ap)
{
    Header *bp, *p;

    bp = (Header *) ap - 1;  /* покажчик на заголовок блока */
    for (p = freep; !(bp > p && bp < p->s.ptr); p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break;    /* звільнений блок напочатку чи вкінці */

    if (bp + bp->size == p->s.ptr) { /* об'єднати до верхнього nbr */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    } else
        bp->s.ptr = p->s.ptr;
    if (p + p->size == bp) {    /* об'єднати до нижнього nbr */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    } else
        p->s.ptr = bp;
    freep = p;
}
```

Незважаючи на те, що операція розподілу пам'яті за визначенням машинозалежна, вищенаведений код демонструє як залежність від машини може бути контрольована та обмежена дуже маленькою частиною програми. Використання `typedef` та `union` (сполук) допомагає з вирівнюванням (за умови, що `sbrk` забезпечить належним покажчиком). Зведення влаштовують, щоб перетворення покажчиків були явними, і навіть справляються з погано розробленим інтерфейсом. Хоча деталі, оговорені тут, стосуються розподілу пам'яті, загальний підхід може застосовуватись так само для інших ситуацій.

Вправа 8-6. Функція стандартної бібліотеки `calloc(n, size)` повертає покажчик на `n` об'єктів розміром `size`, з пам'яттю ініційованою до нуля. Напишіть `calloc`, викликаючи `malloc`, або модифікуючи його.

Вправа 8-7. `malloc` приймає запити розміру без перевірки їхньої вірогідності; `free` вірить в те, що блок, який вона звільняє містить чинне поле `size`. Вдоскональте ці функції, щоб вони докладали більше зусиль для перевірки на помилки.

Вправа 8-8. Напишіть функцію `bfree(p,n)`, яка би звільняла довільний блок `p` на `n` символів, передаючи їх спискові вільної пам'яті, який підтримує `malloc` і `free`. За допомогою `bfree` користувач могтиме коли-завгодно додати статичний чи зовнішній масив до списку вільної пам'яті.

* * *

Додаток A

Додаток A: Довідковий посібник

A.1 A.1 Введення

У цьому посібнику описується мова C як зазначено чернеткою, поданою ANSI 31-го жовтня 1988-го року, для схвалення як «Американського стандарту інформаційних систем — мова програмування C, X3.159-1989». Цей посібник являє собою інтерпретацією висунутого стандарту, а не самим стандартом, хоч було відведено багато уваги, щоб зробити його надійним керівництвом до мови.

Здебільшого, цей документ слідує основним рисам стандарту, який в свою чергу слідує викладу першого видання цієї книжки, хоча їхня організація трохи відрізняється. За винятком зміни назв декількох похідних і відмови від формалізації визначень лексичних зворотів або препроцесора, подана тут граматика мови еквівалентна стандартові.

> У цьому посібнику, коментарі розміщено з відступом і надруковано меншим шрифтом, як от це. Часто, ці коментарі підкреслюють як Стандартна C ANSI відрізняється від мови, визначеній у першому виданні цієї книжки, або від змін, привнесеними подальшими компіляторами.

A.2 A.2 Лексичні умовності

Програма складається з однієї або більше об'єктів перекладу, збережених у файлах. Вони перекладаються в декілька етапів, описаних в частині A.12. Перший етап — це низькорівневі лексичні перетворення, слідуючи директивам у рядках, що починаються зі знака #, які здійснюють визначення макросів та їхнє розкриття. Коли попередня обробка, описана в частині A.12, завершилась, програма розбивається на ряд лексем.

A.2.1 A.2.1 Лексеми

Існує шість класів лексем: ідентифікатори, ключові слова, константи, ланцюжкові літерали, оператори та інші розділювачі. Пробіли, горизонтальна та вертикальна табу-

ляція, знаки нового рядка, знаки подання сторінки та коментарі, як описано нижче, (за загальним визначенням — «знаки пропуску») ігноруються за винятком коли це окремі лексеми. Знаки пропуску необхідні для відділення, у протилежному випадку суміжних, ідентифікаторів, ключових слів і констант.

Якщо потік вводу поділено на лексеми до певного знака, то наступною лексемою буде найдовший ланцюжок, що може скласти лексему.

A.2.2 A.2.2 Коментарі

Символи /* розпочинають коментар, який завершують */. Коментарі не гніздуються, вони не повинні з'являтися посередині ланцюжка символьного літералу.

A.2.3 A.2.3 Ідентифікатори

Ідентифікатор - це послідовність з літер і цифр. Першим символом повинна стояти літера; твердий пробіл _ вважається літерою. Літери верхнього та нижнього регістру вважаються різними. Ідентифікатори можуть бути будь-якої довжини, а для внутрішніх ідентифікаторів, принаймні перші 31 знаки — значимі; у деякий реалізаціях, це число може бути більшим. Внутрішні ідентифікатори можуть складатися з назв макросів препроцесора, і будь-яких імен, що не пов'язані зовнішньо (Розділ A.11.2). На ідентифікатори із зовнішніми зв'язками накладається більше обмежень: деякі реалізації можуть розглядати тільки перші шість знаків, як значимі, і можуть ігнорувати регістрові відмінності літер.

A.2.4 A.2.4 Ключові слова

Наступні ідентифікатори зарезервовано для використання, як ключові слова, і не вживаються інакше:

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

Деякі реалізації також мають зарезервованими слова fortran та asm.

Ключові слова const, signed та volatile є новими, привнесеними стандартом ANSI; enum та void з'явилися після першого видання книжки, але є загальновживаними; entry, колись зарезервоване, але не використане, більше не вважається зарезервованим.

A.2.5 A.2.5 Константи

Існує декілька типів констант. Кожна належить до якогось типу даних; Розділ A.4.2 розгляне базові типи:

константа: цілочисельна-константа символьна-константа дробова-константа константа-енумератор

A.2.5.1 Цілочисельні константи

Цілочисельна константа, яка складається з ряду цифр, вважається вісімковою, якщо починається з 0 (цифра нуль), і десятковою у протилежному випадку. Вісімкові константи не можуть містити цифри 8 та 9. Ряд цифр з попереднім 0x або 0X (нулем та ікс) вважається шістнадцятковим числом, і може включати літери від a або A до f або F, що позначають значення від 10 до 15.

Цілочисельній константі може передувати суфікс u або U, щоб вказати, що вона беззнакова (unsigned). Так само, суфікс l або L вказує на те, що вона довга (long).

Тип цілочисельної константи залежить від її форми, значення та суфікса. (Дивіться Розділ A.4 стосовно обговорення типів.) Якщо суфікса в неї немає і вона є десятковою, тоді тип її складатиме найменший з цих типів, здатний її представити: int, long int, unsigned long int. Якщо суфікса в неї немає, вона — вісімкова або шістнадцяткова, тип її відповідатиме найпершому можливому з наступних: int, unsigned int, long int, unsigned long int. Якщо вона має суфікс u або U, тоді — unsigned int, unsigned long int. Якщо вона має суфікс l або L, тоді — long int, unsigned long int. Якщо цілочисельна константа має суфікс UL, тоді її тип становитиме unsigned long.

Детальний опис типів цілих тут значно переважає наведений у першому виданні, в якому великі цілочисельні константи просто ставали long. Суфікс U — новий.

A.2.5.2 Символьні константи

Символьна константа — це послідовність з одного або більше символів, включених в одинарні лапки, як от 'x'. Значення символьної константи з одним символом відповідає числовому значенню символу в машинному наборі символів під час виконання програми. Значення багатосимвольної константи залежить від реалізації.

Символьні константи не містять символу ' або знаків нового рядка; для представлення їх, а також деяких інших символів, можна скористатися з наступних екранованих послідовностей: знак нового рядка NL(LF) `\n` зворотня похила риска \ `\\` горизонтальна табуляція HT `\t` знак запитання ? `\?` вертикальна табуляція VT `\v` одинарна лапка ' `\'` реверс BS `\b` подвійна лапка ¨ `\¨` повернення каретки CR `\r` вісімкове число ooo `\ooo` зміна сторінки FF `\f` шістнадцяткове число hh `\xhh` звуковий сигнал BEL `\a`

Екранована послідовність \ooo складається зі зворотньої похилої, за якою слідують 1, 2 або три вісімкові цифри, які вказують числове значення бажаного символу. Поширеним прикладом такої конструкції служить \0 (без додаткових цифр), який вказує на нульовий символ (NUL). Екранована послідовність \xhh складається зі зворотньої похилої та літери x, за якими слідують шістнадцяткові цифри, які вказують

числове значення бажаного символу. Обмеження кількості шістнадцяткових цифр не існує, але поводження — невизначене, якщо отримане в наслідок значення символу перевищує значення найбільшого.

Переклад: Віталій Цибуляк[1]
Остання зміна: 22.04.2012

```
$Id: knr-ua,v 1.5 2011/11/28 10:45:08 nabis Exp $
```

[1]See URL mailto:uatech@meta.ua