5. Покажчики та масиви

***Покажчик*** - це змінна, яка містить адресу іншої змінної. покажчики дуже широко використовуються в мові "C". Це відбувається частково тому, що іноді вони дають єдину можливість висловити потрібну дію, а також тому, що вони зазвичай ведуть до більш компактних і ефективних програмх, ніж ті, які можуть бути отримані іншими способами.

Покажчики зазвичай змішують в одну купу з операторами goto, характеризуючи їх як чудовий спосіб написання програм, які неможливо зрозуміти. Це безумовно справедливо, якщо покажчики використовуються безтурботно; дуже просто ввести покажчики, які вказують на щось абсолютно несподіване. Однак, при певній дисципліні, використання покажчиків допомагає досягти ясності і простоти. Саме цей аспект ми спробуємо тут проілюструвати.

5.1. Покажчики та адреси

Так як покажчик містить адресу об'єкта, це дає можливість "непрямого" доступу до цього об'єкта через покажчик. Припустимо, що х - змінна, наприклад, типу int, а рх - покажчик, створений таким собі ще не зазначеним способом. Унарна операція & видає адресу об'єкта, так що оператор

рх = &х;

привласнює адресу х змінній рх; кажуть, що рх "вказує" на х. Операція & застосовується тільки до змінних і елементів масиву, конструкції виду & (х-1) і & 3 є незаконними. Не можна також отримати адресу реєстрової змінної.

Унарна операція \* розглядає свій операнд як адресу кінцевої мети і звертається за цією адресою, щоб витягти вміст. Отже, якщо y теж має тип int, то

y = \*рх;

присвоює y вміст того, на що вказує рх. Так послідовність

рх = &х;

y = \*рх;

присвоює y те ж саме значення, що і оператор

y = x;

змінні, що беруть участь у всьому цьому необхідно описати:

int x, y; int \*px;

з описом для x і y ми вже неодноразово зустрічалися. Опис покажчика

int \*px;

є новим і має розглядатися як мнемонічне; воно говорить, що комбінація \* px має тип int. Це означає, що якщо px з'являється в контексті \* px, то це еквівалентно змінній типу int. Фактично синтаксис опису змінної імітує синтаксис виразів, в яких ця змінна може з'являтися. Це зауваження корисно в усіх випадках, пов'язаних зі складними описами. Наприклад,

double atof(), \*dp;

каже, що atof () і \* dp мають у виразах значення типу double.  
Ви повинні також зауважити, що з цього опису випливає, що покажчик може вказувати тільки на певний вид об'єктів.  
Покажчики можуть входити у вираз. Наприклад, якщо px вказує на ціле x, то \* px може з'являтися в будь-якому контексті, де може зустрітися x. Так оператор

y = \*px + 1

присвоює y значення, на 1 більше значення x;

printf("%d\n", \*px)

друкує поточне значення x ;

d = sqrt((double) \*px)

отримує в d квадратний корінь з x, причому до передачі функції sqrt значення x перетворюється до типу double. (Дивись"лекцію №2" ).

У виразах виду

y = \*px + 1

унарні операції \* і & пов'язані зі своїм операндом міцніше, ніж арифметичні операції, так що такий вислів бере то значення, на яке вказує px, додає 1 і привласнює результат змінної y. Ми незабаром повернемося до того, що може означати вираз

y = \*(px + 1)

Посилання на покажчики можуть з'являтися і в лівій частині присвоювання. Якщо px вказує на x, то

\*px = 0

вважає x рівним нулю, а

\*px += 1

збільшує його на одиницю, як і вираз

(\*px)++

Круглі дужки в останньому прикладі необхідні; якщо їх опустити, то оскільки унарні операції, подібні \* і ++, виконуються справа наліво, цей вислів збільшить px, а не ту змінну, на яку він вказує.  
І нарешті, так як покажчики є змінними, то з ними можна звертатися, як і з іншими змінними. Якщо py - інший покажчик на змінну типу int, то

py = px

копіює вміст px в py, в результаті чого py вказує на те ж, що і px.

#### 5.2. Покажчики та аргументи функцій

Так як в "с" передача аргументів функцій здійснюється "за значенням", викликана процедура не має безпосередньої можливості змінити змінну з програми що викликає. Що ж робити, якщо вам дійсно треба змінити аргумент? Наприклад, програма сортування захотіла б поміняти два порушуючі порядок елемента за допомогою функції з ім'ям swap. Для цього недостатньо написати

swap(a, b);

визначивши функцію swap при цьому наступним чином:

swap(x, y) /\* wrong \*/

int x, y;

{

int temp;

temp = x;

x = y;

y = temp;

}

через виклик за значенням swap не може впливати на аргументи a і b в викликає функції.

На щастя, все ж є можливість отримати бажаний ефект. Виклиуаюча програма передає покажчики що підлягають зміні значень: swap (& a, & b); так як операція & видає адресу змінної, то & a є покажчиком на a. У самій swap аргументи описуються як покажчики і доступ до фактичних операндам здійснюється через них.

swap(px, py) /\* interchange \*px and \*py \*/

int \*px, \*py;

{

int temp;

temp = \*px;

\*px = \*py;

\*py = temp;

}

Покажчики в якості аргументів зазвичай використовуються у функціях, які повинні повертати більше одного значення. (Можна сказати, що swap повертає два значення, нові значення її аргументів). Як приклад розглянемо функцію getint, яка здійснює перетворення вступників у вільному форматі даних, розділяючи потік символів на цілі значення, по одному цілому за одне звернення. Функція getint повинна повертати або знайдене значення, або ознака кінця файлу, якщо вхідні дані повністю вичерпані. Ці значення повинні повертатися як окремі об'єкти, яке б значення ні використовувалося для EOF, навіть якщо це значення вводиться цілого.

Одне з рішень, що грунтується на описуваній в "лекції №7" функції введення scanf, полягає в тому, щоб при виході на кінець файлу getint повертала EOF як значення функції; будь-яке інше повернене значення говорить про знаходження нормального цілого. Чисельне ж значення знайденого цілого повертається через аргумент, який повинен бути покажчиком цілого. Ця організація розділяє статус кінця файлу і чисельні значення.

Наступний цикл заповнює масив цілими за допомогою звернень до функції

getint:

int n, v, array[size];

for (n = 0; n < size && getint(&v) != EOF; n++)

array[n] = v;

В результаті кожного звернення v стає рівним наступному цілому значенню, знайденому у вхідних даних. Зверніть увагу, що в якості аргументу getint необхідно вказати & v а не v. Використання просто v швидше за все призведе до помилки адресації, оскільки getint вважає, що вона працює саме з покажчиком.

Сама getint є очевидною модифікацією написаної нами раніше функції atoi:

getint(pn) /\* get next integer from input \*/

int \*pn;

{

int c,sign;

while ((c = getch()) == ' ' || c == '\n'

|| c == '\t'); /\* skip white space \*/

sign = 1;

if (c == '+' || c == '-') { /\* record

sign \*/

sign = (c == '+') ? 1 : -1;

c = getch();

}

for (\*pn = 0; c >= '0' && c <= '9'; c = getch())

\*pn = 10 \* \*pn + c - '0';

\*pn \*= sign;

if (c != EOF)

ungetch(c);

return(c);

}

Вираз \* pn використовується всюди в getint як звичайна змінна типу int. Ми також використовували функції getch і ungetch (описані в "лекції №4"), так що один зайвий символ, який доводиться зчитувати, може бути поміщений назад у введення.

**Вправа 5-1**

Напишіть функцію getfloat, аналог getint для чисел з плаваючою крапкою. Який тип має повертати getfloat як значення функції?

#### 5.3. Покажчики та масиви

У мові "C" існує сильний взаємозв'язок між покажчиками і масивами, настільки сильний, що покажчики і масиви дійсно слід розглядати одночасно. Будь-яку операцію, яку можна виконати за допомогою індексів масиву, можна зробити і за допомогою покажчиків. Варіант з покажчиками зазвичай виявляється більш швидким, але і більш важким для безпосереднього розуміння, принаймні для початківця. Опис

int a[10]

визначає масив розміру 10, тобто набір з 10 послідовних об'єктів, званих a [0], a [1], ..., a [9]. Запис a [i] відповідає елементу масиву через i позицій від початку. Якщо pa - покажчик цілого, описаний як

int \*pa

то присвоєння

pa = &a[0]

призводить до того, що pa вказує на нульовий елемент масиву a; це означає, що pa містить адресу елемента a [0].

Тепер присвоювання

x = \*pa

буде копіювати вміст a [0] в x.

Якщо pa вказує на деякий певний елемент масиву a, то за визначенням pa + 1 вказує на наступний елемент, і взагалі pa-i вказує на елемент, що стоїть на i позицій до елемента, що вказується pa, а pa + i на елемент, що стоїть на i позицій після. Таким чином, якщо pa вказує на a [0], то

\*(pa+1)

посилається на вміст a [1], pa + i - адреса a [i], а \* (pa + i) - вміст a [i].

Ці зауваження справедливі незалежно від типу змінних в масиві a. Суть визначення "додавання 1 до покажчика", а також його поширення на всю арифметику покажчиків, полягає в тому, що приріст масштабується розміром пам'яті, займаної об'єктом, на який вказує покажчик. Таким чином, i в pa + i перед додаванням множиться на розмір об'єктів, на які вказує pa.

Очевидно існує дуже тісна відповідність між індексацією і арифметикою покажчиків. в дійсності компілятор перетворює посилання на масив в покажчик на початок масиву. В результаті цього ім'я масиву є вказівним виразом. Звідси випливає кілька вельми корисних наслідків. Так як ім'я масиву є синонімом місця розташування його нульового елемента, то присвоювання pa = & a [0] можна записати як

pa = a

Ще більш дивним, принаймні на перший погляд, здається той факт, що посилання на a [i] можна записати у вигляді \* (a + i). При аналізуванні вираження a [i] в ​​мові "C" воно негайно перетвориться до виду \* (a + i); ці дві форми абсолютно еквівалентні. Якщо застосувати операцію & до обох частин такого співвідношення еквівалентності, то ми отримаємо, що & a [i] і a + i теж ідентичні: a + i - адреса i -го елемента від початку a. З іншого боку, якщо pa є покажчиком, то у виразах його можна використовувати з індексом: pa [i] ідентично \* (pa + i). Коротше, будь-який вираз, що включає масиви і індекси, може бути записано через покажчики і зміщення і навпаки, причому навіть в одному і тому ж затвердження.

Є одна відмінність між ім'ям масиву і покажчиком, яке необхідно мати на увазі. покажчик є змінною, так що операції pa = a і pa ++ мають сенс. Але ім'я масиву є константою, а не змінною: конструкції типу a = pa або a ++, або p = & a будуть незаконними.

Коли ім'я масиву передається функції, то насправді їй передається розташування початку цього масиву. Усередині викликаної функції такий аргумент є такою самою змінною, як і будь-яка інша, так що ім'я масиву як аргумент дійсно є покажчиком, тобто змінної, що містить адресу. Ми можемо використовувати цю обставину для написання нового варіанта функції strlen, що обчислює довжину рядка.

strlen(s) /\* return length of string s \*/

char \*s;

{

int n;

for (n = 0; \*s != '\0'; s++)

n++;

return(n);

}

Операція збільшення s абсолютно законна, оскільки ця змінна є покажчиком; s ++ ніяк не впливає на символьний рядок в звернулася до strlen функції, а тільки збільшує локальну для функції strlen копію адреси. Опис формальних параметрів у визначенні функції у вигляді

char s[]; char \*s;

абсолютно еквівалентні; який вид опису слід віддати перевагу, визначається в значній мірі тим, які вислови будуть використані при написанні функції. Якщо функції передається ім'я масиву, то в залежності від того, що зручніше, можна вважати, що функція оперує або з масивом, або з покажчиком, і діяти далі відповідним чином. Можна навіть використовувати обидва види операцій, якщо це здається доречним і ясним.

Можна передати функції частину масиву, якщо задати в якості аргументу покажчик початку підмасиву. Наприклад, якщо a - масив, то як

f(&a[2])

так і

f(a+2)

передають функції f адреса елемента a [2], тому що і & a [2], і a + 2 є вказівними виразами, що посилаються на третій елемент a. Усередині функції f опису аргументів можуть бути присутніми у вигляді:

f(arr) int arr[];

{

...

}

або

f(arr) int \*arr;

{

...

}

Що стосується функції f, то той факт, що її аргумент насправді посилається до частини більшого масиву, не має для неї ніяких наслідків.

#### 5.4. Адресна арифметика

Якщо p є покажчиком, то яким би не був сорт об'єкта, на який він вказує, операція p ++ збільшує p так, що він вказує на наступний елемент набору цих об'єктів, а операція p + = i збільшує p так, щоб він вказував на елемент, віддалений на i елементів від поточного елемента. Ці та аналогічні конструкції представляють собою найпростіші і найпоширеніші форми арифметики покажчиків або адресної арифметики.

Мова "C" послідовнаі постійна в своєму підході до адресної арифметики; об'єднання в одне ціле покажчиків, масивів та адресної арифметики є однією з найбільш сильних сторін мови. Давайте проілюструємо деякі з відповідних можливостей мови на прикладі елементарної (але корисної, незважаючи на свою простоту) програми розподілу пам'яті.

Є дві функції: функція alloc (n) повертає в якості свого значення покажчик p, який вказує на першу з n послідовних символьних позицій, які можуть бути використані викликає функцію alloc програмою для зберігання символів; функція free (p) звільняє придбану таким чином пам'ять, так що її в подальшому можна знову використовувати. Програма є "елементарною", тому що звернення до free повинні проводитися в порядку, зворотному тому, в якому проводилися звернення до alloc.

Таким чином, керована функціями alloc і free пам'ять є стеком або списком, в якому останній вводиться елемент витягується першим. Стандартна бібліотека мови "C" містить аналогічні функції, що не мають таких обмежень, і, крім того, в "лекції №8" ми наведемо поліпшені варіанти. Тим часом, однак, для багатьох додатків потрібна тільки тривіальна функція alloc для розподілу невеликих ділянок пам'яті невідомих заздалегідь розмірів в непередбачувані моменти часу.

Найпростіша реалізація полягає в тому, щоб функція роздавала відрізки великого символьного масиву, якому ми присвоїли ім'я allocbuf. Цей масив є власністю функцій alloc і free. Так як вони працюють з покажчиками, а не з індексами масиву, ніякій іншій функції не потрібно знати ім'я цього масиву. Він може бути описаний як зовнішній статичний, тобто він буде локальним по відношенню до вихідного файлу, що містить alloc і free, і невидимим за його межами. При практичній реалізації цей масив може навіть не мати імені; замість цього він може бути отриманий в результаті запиту до операційної системи на покажчик деякого неіменованого блоку пам'яті.

Іншою необхідною інформацією є те, яка частина масиву allocbuf вже використана. Ми користуємося покажчиком першого вільного елемента, названим allocp. Коли до функції alloc звертаються за виділенням n символів, то вона перевіряє, чи достатньо залишилося для цього місця в allocbuf. Якщо досить, то alloc повертає поточне значення allocp (тобто початок вільного блоку), потім збільшує його на n, з тим щоб він вказував на наступну вільну область. функція free (p) просто вважає allocp рівним p за умови, що p вказує на позицію всередині allocbuf.

#define null 0 /\* pointer value for error report \*/

#define allocsize 1000 /\* size of available space \*/

static char allocbuf[allocsize];/\* storage for alloc \*/

static char \*allocp = allocbuf; /\* next free position \*/

char \*alloc(n) /\* return pointer to n characters \*/

int n;

(

if (allocp + n <= allocbuf + allocsize) {

allocp += n;

return(allocp - n); /\* old p \*/

} else /\* not enough room \*/

return(null);

)

free(p) /\* free storage pointed by p \*/

char \*p;

(

if (p >= allocbuf && p < allocbuf + allocsize)

allocp = p;

)

Дамо деякі пояснення. Взагалі кажучи, покажчик може бути ініціалізованим так само, як і будь-яка інша змінна, хоча зазвичай єдиними осмисленими значеннями є null (це обговорюється нижче) або вираз, що включає адреси раніше визначених даних відповідного типу.  
Опис

static char \*allocp = allocbuf;

визначає allocp як покажчик на символи і ініціалізує його так, щоб він вказував на allocbuf, тобто на першу вільну позицію при початку роботи програми. Так як ім'я масиву є адресою його нульового елемента, то це можна було б записати у вигляді

static char \*allocp = &allocbuf[0];

використовуйте той запис, який вам здається більш природним. За допомогою перевірки

if (allocp + n <= allocbuf + allocsize)

з'ясовується, чи залишилося достатньо місця, щоб задовольнити запит на n символів. Якщо достатньо, то нове значення allocp не вказуватиме далі, ніж на останню позицію allocbuf. Якщо запит може бути задоволений, то alloc повертає звичайний покажчик (зверніть увагу на опис самої функції). Якщо ж ні, то alloc повинна повернути деяку ознаку, що говорить про те, що більше місця не залишилося. У мові "C" гарантується, що жоден правильний покажчик даних не може мати значення нуль, так що повернення нуля може служити в якості сигналу про ненормальний подію, в даному випадку про відсутність місця. Ми, однак, замість нуля пишемо null, з тим щоб більш ясно показати, що це спеціальне значення покажчика. Взагалі кажучи, цілі не можуть осмислено присвоюватися вказівниками, а нуль - це особливий випадок.  
Перевірки виду

if (allocp + n <= allocbuf + aloocsize) и

if (p >= allocbuf && p < allocbuf + allocsize)

демонструють кілька важливих аспектів арифметики покажчиків. По-перше, за певних умов покажчики можна порівнювати. Якщо p і q вказують на елементи одного і того ж масиву, то такі відносини, як <,> = і т.д., працюють належним чином. Наприклад,

p < q

істинно, якщо p вказує на більш ранній елемент масиву, ніж q. Відносини == і! = теж працюють. Будь-покажчик можна осмисленим чином порівняти на рівність чи нерівність з null. Але ні за що не можна ручатися, якщо ви використовуєте порівняння при роботі з покажчиками, що вказують на різні масиви. Якщо вам пощастить, то на всіх машинах ви отримаєте очевидну нісенітницю. Якщо ж ні, то ваша програма буде правильно працювати на одній машині і давати незбагненні результати на інший.

По-друге, як ми вже бачили, покажчик і ціле можна додавати і віднімати. Конструкція

p + n

має на увазі n -ий об'єкт за тим, на який p вказує зараз. Це справедливо незалежно від того, на який вид об'єктів p повинен вказувати; компілятор сам масштабує n відповідно до обумовлених з опису p розміром об'єктів, що вказуються за допомогою p. Наприклад, на PDP-11 масштабуючий множник дорівнює 1 для char, 2 для int і short, 4 для long і float і 8 для double.

Віднімання покажчиків теж можливо: якщо p і q вказують на елементи одного і того ж масиву, то p-q - кількість елементів між p і q. Цей факт можна використовувати для написання ще одного варіанту функції

strlen:

strlen(s) /\* return length of string s \*/

char \*s;

{

char \*p = s;

while (\*p != '\0')

p++;

return(p-s);

}

При описі покажчик p в цій функції ініціалізований за допомогою рядка s, в результаті чого він вказує на перший символ рядка. У циклі while по черзі перевіряється кожен символ до тих пір, поки не з'явиться символ кінця рядка \ 0. Так як значення \ 0 дорівнює нулю, а while тільки з'ясовує, чи має вираз в ньому значення 0, то в даному випадку явну перевірку можна опустити. Такі цикли часто записують у вигляді

while (\*p)

p++;

Так як p вказує на символи, то оператор p ++ пересуває p щоразу так, щоб він вказував на наступний символ. В результаті p-s дає число переглянутих символів, тобто довжину рядка. Арифметика покажчиків послідовна: якби ми мали справу зі змінними типу float, які займають більше пам'яті, ніж змінні типу char, і якби p був покажчиком на float, то оператор p ++ пересунув би p на наступне float.

Таким чином, ми могли б написати інший варіант функції alloc, що розподіляє пам'ять для float, замість char, просто замінивши всюди в alloc і free описатель char на float. Всі дії з покажчиками автоматично враховують розмір об'єктів, на які вони вказують, так що більше нічого міняти не треба.

За винятком згаданих вище операцій (додавання і віднімання покажчика і цілого, віднімання і порівняння двох покажчиків), вся інша арифметика покажчиків є незаконною. Заборонено складати два покажчика, множити, ділити, зрушувати або маскувати їх, а також додавати до них змінні типу float або double.

#### 5.5. Покажчики символів та функції

Рядкова константа, така, як наприклад,

"i am a string"

є масивом символів. Компілятор завершує внутрішнє уявлення такого масиву символом \ 0, так що програми можуть знаходити його кінець. Таким чином, довжина масиву в пам'яті виявляється на одиницю більше числа символів між подвійними лапками.  
Мабуть найчастіше рядкові константи з'являються як аргументи функцій, як, наприклад, в

printf ("hello, world\n");

коли символьний рядок, подібний до цього, з'являється в програмі, то доступ до нього здійснюється за допомогою покажчика символів; функція printf фактично отримує покажчик символьного масиву.  
Звичайно, символьні масиви не зобов'язані бути тільки аргументами функцій. Якщо описати message як

char \*message;

то в результаті оператора

message = "now is the time";

змінна message стане покажчиком на фактичний масив символів. Це не копіювання рядка; тут беруть участь тільки покажчики. У мові "C" не передбачені будь-які операції для обробки всього рядка символів як цілого.  
Ми проілюструємо інші аспекти покажчиків і масивів, розбираючи дві корисні функції зі стандартної бібліотеки вводу-виводу, яка буде розглянута в "лекції №7".  
Перша функція - це strcpy (s, t), яка копіює рядок t в рядок s. Аргументи написані саме в цьому порядку за аналогією з операцією привласнення, коли для того, щоб привласнити t до s зазвичай пишуть

s = t

спочатку наведемо версію з масивами:

strcpy(s, t) /\* copy t to s \*/

char s[], t[];

{

int i;

i = 0;

while ((s[i] = t[i]) != '\0')

i++;

}

Для зіставлення нижче дається варіант strcpy з покажчиками.

strcpy(s, t) /\* copy t to s; pointer version 1 \*/

char \*s, \*t;

{

while ((\*s = \*t) != '\0') {

s++;

t++;

}

}

Так як аргументи передаються за значенням, функція strcpy може використовувати s і t так, як вона побажає. Тут вони з зручністю покладаються покажчиками, які пересуваються вздовж масивів, по одному символу за крок, поки не буде скопійований в s завершальний в t символ \ 0.

На практиці функція strcpy була б записана не так, як ми показали вище. Ось друга можливість:

strcpy(s, t) /\* copy t to s; pointer version 2 \*/

char \*s, \*t;

{

while ((\*s++ = \*t++) != '\0')

;

}

Тут збільшення s і t внесено в перевірочну частину. Значним \* t ++ є символ, на який вказував t до збільшення; Постфіксна операція ++ не змінює t, поки цей символ НЕ буде витягнутий. Точно так же цей символ поміщається в стару позицію s, до того як s буде збільшено. Кінцевий результат полягає в тому, що всі символи, включаючи завершальний \ 0, копіюються з t в s.

І як останнє скорочення ми знову відзначимо, що порівняння з \ 0 є зайвим, так що функцію можна записати у вигляді

strcpy(s, t) /\* copy t to s; pointer version 3 \*/

char \*s, \*t;

{

while (\*s++ = \*t++)

;

}

хоча з першого погляду цей запис може здатися загадкової, вона дає значну зручність. Цією ідіомою слід опанувати вже хоча б тому, що ви з нею будете часто зустрічатися в "C" - програмах.

Друга функція - strcmp (s, t), яка порівнює символьні рядки s та t, повертаючи негативне, нульове або позитивне значення відповідно до того, менше, дорівнює або більше лексикографічно s, ніж t. Значення, що повертається виходить в результаті віднімання символів з першої позиції, в якій s і t не збігаються.

strcmp(s, t) /\* return <0 if s<t, 0 if s==t, >0 if s>t \*/

char s[], t[];

{

int i;

i = 0;

while (s[i] == t[i])

if (s[i++] == '\0')

return(0);

return(s[i]-t[i]);

}

Ось версія strcmp з покажчиками:

strcmp(s, t) /\* return <0 if s<t, 0 if s==t, >0 if s>t \*/

char \*s, \*t;

{

for ( ; \*s == \*t; s++, t++)

if (\*s == '\0')

return(0);

return(\*s-\*t);

}

так як ++ і - можуть бути як постфіксними, так і префіксними операціями, зустрічаються інші комбінації \* і ++ і -, хоча і менш часто.

Наприклад

\*++p

збільшує p до вилучення символу, на який вказує p, а

\*--p

Спочатку зменшує p.

**Вправа 5-2**

Напишіть варіант з покажчиками функції strcat з "лекції №2" : strcat(s, t) копіює рядок t в кінець s.

**Вправа 5-3**

Напишіть макрос для strcpy.

**Вправа 5-4**

Перепишіть підходящі програми з попередніх лекцій і вправ, використовуючи покажчики замість індексації масивів. Хороші можливості для цього надають функції getline / "лекції 1" і "№4" /, atoi, itoa та їх варіанти / "лекція №2" , "№3" і"№4" /, reverse / "лекція №3" /, index і getop / "лекція №4" /.

#### 5.6. Покажчики - не цілі

Ви, можливо, звернули увагу в попередніх "с" - програмах на досить невимушене ставлення до копіювання покажчиків. Загалом це правильно, що на більшості машин покажчик можна привласнити цілому і передати його назад, не змінивши його; при цьому не відбувається ніякого масштабування або перетворення і жоден біт не втрачається. На жаль, це веде до вільного поводження з функціями, які повертають покажчики, які потім просто передаються іншим функціям, - необхідний опис покажчиків часто опускається. Розглянемо, наприклад, функцію strsave (s), яка копіює рядок s в деякий місце для зберігання, що виділяється за допомогою звернення до функції alloc, і повертає покажчик на це місце. Правильно вона повинна бути записана так:

char \*strsave(s) /\* save string s somewhere \*/

char \*s;

{

char \*p, \*alloc();

if ((p = alloc(strlen(s)+1)) != null)

strcpy(p, s);

return(p);

}

на практиці існує сильне прагнення опускати опис:

\*strsave(s) /\* save string s somewhere \*/

{

char \*p;

if ((p = alloc(strlen(s)+1)) != null)

strcpy(p, s);

return(p);

}

Ця програма буде правильно працювати на багатьох машинах, тому що за замовчуванням функції і аргументи мають тип int, а покажчик і ціле зазвичай можна безпечно пересилати туди і назад. Однак такий стиль програмування в своїй істоті є ризикованим, оскільки залежить від деталей реалізації та архітектури машини і може привести до неправильних результатів на конкретному використовуваному вами компіляторі. Розумніше всюди використовувати повні описи. (Отладочная програма lint попередить про таких конструкціях, якщо вони по необережності все ж з'являться).

#### 5.7. Багатовимірні масиви

У мові "C" передбачені прямокутні багатовимірні масиви, хоча на практиці існує тенденція до їх значно більше рідкого використання в порівнянні з масивами вказівників. У цьому розділі ми розглянемо деякі їх властивості.

Розглянемо задачу перетворення дня місяця в день року і навпаки. Наприклад, 1-е березня є 60-м днем не високосного року і 61-м днем високосного року. Давайте введемо дві функції для виконання цих перетворень: day\_of\_year перетворює місяць і день в день року, а month\_day перетворює день року в місяць і день. Так як ця остання функція повертає два значення, то аргументи місяці і дні повинні бути покажчиками:

month\_day(1977, 60, &m, &d)

Вважає m рівним 3 і d рівним 1 (1-е березня).

Обидві ці функції потребують однієї і тієї ж інформаційної таблиці, що вказує число днів в кожному місяці. Так як число днів у місяці в високосному і в невисокосному році відрізняється, то простіше уявити їх у вигляді двох рядків двовимірного масиву, ніж намагатися простежувати під час обчислень, що саме відбувається в лютому. Ось цей масив і виконують ці перетворення функції:

static int day\_tab[2][13] = {

(0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31),

(1, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31)

};

day\_of\_year(year, month, day) /\* set day of year \*/

int year, month, day; /\* from month & day \*/

{

int i, leap;

leap = year%4 == 0 && year%100 != 0 || year%400 == 0;

for (i = 1; i < month; i++)

day += day\_tab[leap][i];

return(day);

{

month\_day(year, yearday, pmonth, pday) /\*set month,day \*/

int year, yearday, \*pmonth, \*pday; /\* from day of year \*/

{

leap = year%4 == 0 && year%100 != 0 || year%400 == 0;

for (i = 1; yearday > day\_tab[leap][i]; i++)

yearday -= day\_tab[leap][i];

\*pmonth = i;

\*pday = yearday;

}

Масив day\_tab повинен бути зовнішнім як для day\_of\_year, так і для month\_day, оскільки він використовується обома цими функціями.

Масив day\_tab є першим двовимірним масивом, з яким ми маємо справу. За визначенням в "C" двовимірний масив по суті є одновимірним масивом, кожен елемент якого є масивом. Тому індекси записуються як

day\_tab[i][j] а не day\_tab [i, j]

як в більшості мов. В іншому з двовимірними масивами можна в основному звертатися таким же чином, як в інших мовах. Елементи зберігаються по рядках, тобто при зверненні до елементів в порядку їх розміщення в пам'яті швидше за все змінюється самий правий індекс.

Масив ініціалізується за допомогою списку початкових значень, укладених у фігурні дужки; кожен рядок двовимірного масиву ініціалізується відповідним підсписком. Ми помістили в початок масиву day\_tab стовпець з нулів для того, щоб номери місяців змінювалися природним чином від 1 до 12, а не від 0 до 11. Так як за економію пам'яті у нас поки не нагороджують, такий спосіб простіше, ніж підгонка індексів.

Якщо двовимірний масив передається функції, то опис відповідного аргументу функції повинно містити кількість стовпців; кількість рядків несуттєва, оскільки, як і раніше, фактично передається покажчик. У нашому конкретному випадку це покажчик об'єктів, які є масивами з 13 чисел типу int. Таким чином, якби було потрібно передати масив day\_tab функції f, то опис в f мав би вигляд:

f(day\_tab) int day\_tab[2][13];

{

...

}

Так як кількість рядків є несуттєвою, то опис аргументу в f міг би бути таким:

int day\_tab[][13];

чи таким

int (\*day\_tab)[13];

в якому говориться, що аргумент є покажчиком масиву з 13 цілих. Круглі дужки тут потрібні, тому що квадратні дужки [] мають більш високий рівень старшинства, ніж \*; як ми побачимо в наступному розділі, без круглих дужок

int \*day\_tab[13];

є описом масиву з 13 покажчиків на цілі.

#### 5.8. Масиви покажчиків; покажчики покажчиків

Так як покажчики самі є змінними, то ви цілком могли б очікувати використання масиву покажчиків. Це дійсно так. Ми проілюструємо це написанням програми сортування в алфавітному порядку набору текстових рядків, гранично спрощеного варіанту утиліти sort операційної системи UNIX.

В "лекції №3" ми привели функцію сортування по Шеллу, яка впорядковувала масив цілих. Цей же алгоритм буде працювати і тут, хоча тепер ми будемо мати справу з рядками тексту різної довжини, які, на відміну від цілих, не можна порівнювати або переміщати за допомогою однієї операції. Ми потребуємо в такому поданні даних, яке б дозволяло зручно і ефективно обробляти рядки тексту змінної довжини.

Тут і виникають масиви покажчиків. Якщо підляючі сортуванню рядка зберігаються одна за одною в довгому символьному масиві (керованому, наприклад, функцією alloc), то до кожного рядка можна звернутися за допомогою покажчика на її перший символ. Самі покажчики можна зберігати в масиві. Два рядки можна порівняти, передавши їх покажчики функції strcmp.

Якщо дві розташовані в неправильному порядку рядки повинні бути переставлені, то фактично переставляються покажчики в масиві покажчиків, а не самі тексти рядків. Цим виключаються відразу дві пов'язані проблеми: складного управління пам'яттю і великих додаткових витрат на фактичну перестановку рядків.

Процес сортування включає три кроки:

* читання всіх рядків вводу
* їх сортування
* виведення їх в правильному порядку

Як завжди, краще розділити програму на кілька функцій відповідно до природного розподілом завдання і виділити провідну функцію, яка керує роботою всієї програми. Давайте відкладемо на деякий час розгляд кроку сортування і зосередимося на структурі даних і введення-виведення. Функція, що здійснює введення, повинна отримати символи кожного рядка, запам'ятати їх і побудувати масив покажчиків рядків. Вона повинна також підрахувати число рядків у введенні, так як ця інформація необхідна при сортуванні та виведення.

Так як функція введення в змозі впоратися тільки з кінцевим числом вводяться рядків, в разі занадто великого їх числа вона може повертати деяке число, відмінне від можливого числа рядків, наприклад -1. Функція здійснює висновок, повинна друкувати рядки в тому порядку, в якому вони з'являються в масиві покажчиків.

#define null 0

#define lines 100 /\* max lines to be sorted \*/

main() /\* sort input lines \*/

{

char \*lineptr[lines]; /\*pointers to text lines \*/

int nlines; /\* number of input lines read \*/

if ((nlines = readlines(lineptr, lines)) >= 0) {

sort(lineptr, nlines);

writelines(lineptr, nlines);

}

else

printf("input too big to sort\n");

}

#define maxlen 1000

readlines(lineptr, maxlines) /\* read input lines \*/

char \*lineptr[]; /\* for sorting \*/

int maxlines;

{

int len, nlines;

char \*p, \*alloc(), line[maxlen];

nlines = 0;

while ((len = getline(line, maxlen)) > 0)

if (nlines >= maxlines)

return(-1);

else if ((p = alloc(len)) == null)

return (-1);

else {

line[len-1] = '\0'; /\* zap newline \*/

strcpy(p,line);

lineptr[nlines++] = p;

}

return(nlines);

}

Символ нового рядка в кінці кожного рядка віддаляється, так що він ніяк не впливатиме на порядок, в якому упорядковано рядки.

writelines(lineptr, nlines) /\* write output lines \*/

char \*lineptr[];

int nlines;

{

int i;

for (i = 0; i < nlines; i++)

printf("%s\n", lineptr[i]);

}

Істотно новим в цій програмі є опис

char \*lineptr[lines];

яке повідомляє, що lineptr є масивом з lines елементів, кожен з яких - покажчик на змінні типу char. Це означає, що lineptr [i] - покажчик на символи, а \* lineptr [i] витягує символ.

Так як сам lineptr є масивом, який передається функції writelines, з ним можна звертатися як з покажчиком точно таким же чином, як в наших попередніх прикладах. Тоді останню функцію можна переписати у вигляді:

writelines(lineptr, nlines) /\* write output lines \*/

char \*lineptr[];

int nlines;

{

int i;

while (--nlines >= 0)

printf("%s\n", \*lineptr++);

}

Т ут \* lineptr спочатку вказує на перший рядок; кожне збільшення пересуває покажчик на наступний рядок, в той час як nlines убуває до нуля.

Впоравшись з введенням і виведенням, ми можемо перейти до сортування. Програма сортування по Шеллу з "лекції №3" вимагає дуже невеликих змін: повинні бути модифіковані опису, а операція порівняння виділена в окрему функцію. Основний алгоритм залишається тим же самим, і це дає нам певну впевненість, що він як і раніше буде працювати.

sort(v, n) /\* sort strings v[0] ... v[n-1] \*/

char \*v[]; /\* into increasing order \*/

int n;

{

int gap, i, j;

char \*temp;

for (gap = n/2; gap > 0; gap /= 2)

for (i = gap; i < n; i++)

for (j = i - gap; j >= 0; j -= gap) {

if (strcmp(v[j], v[j+gap]) <= 0)

break;

temp = v[j];

v[j] = v[j+gap];

v[j+gap] = temp;

}

}

Так як кожен окремий елемент масиву v (ім'я формального параметра, відповідного lineptr) є покажчиком на символи, то і temp повинен бути покажчиком на символи, щоб їх було можна копіювати одне в одного. Ми написали цю програму по можливості більш просто з тим, щоб швидше отримати працюючу програму. Вона могла б працювати швидше, якщо, наприклад, вводити рядки безпосередньо в масив, керований функцією readlines, а не копіювати їх в line, а потім в приховане місце за допомогою функції alloc.

Але ми вважаємо, що буде розумніше початковий варіант зробити простішим для розуміння, а про "ефективності" подбати пізніше. Тим не менш, очевидно, спосіб, що дозволяє досягти помітного прискорення роботи програми полягає не у виключенні зайвого копіювання вводяться рядків. Більш ймовірно, що суттєвої різниці можна досягти за рахунок заміни сортування за Шеллі на щось краще, наприклад, на метод швидкого сортування.

В "лекции №1" ми відзначали, що оскільки в циклах while і for перевірка здійснюється до того, як тіло циклу виконається хоча б один раз, ці цикли виявляються зручними для забезпечення належного функціонування програми при граничних значеннях, зокрема, коли введення взагалі немає. Дуже корисно переглянути всі функції програми сортування, розбираючись, що відбувається, якщо вводиться текст відсутній.

**Вправа 5-5**

Перепишіть функцію readlines таким чином, щоб вона поміщала рядки в масив, що надається функцією main, а не в пам'ять, керовану зверненнями до функції alloc. Наскільки швидше стала програма?

#### 5.9. Ініціалізація масивів покажчиків

Розглянемо задачу написання функції month\_name (n), яка повертає покажчик на символьний рядок, що містить ім'я n -го місяця. Це ідеальна завдання для застосування внутрішнього статичного масиву. функція month\_name містить локальний масив символьних рядків і при зверненні до неї повертає покажчик потрібного рядка. Тема цього розділу - як форматувати цей масив імен.

char \*month\_name(n) /\* return name of n-th month \*/

int n;

{

static char \*name[] = {

"illegal month",

"january",

"february",

"march",

"april",

"may",

"jun",

"july",

"august",

"september",

"october",

"november",

"december"

};

return ((n < 1 || n > 12) ? name[0] : name[n]);

}

опис масиву покажчиків на символи name точно таке ж, як аналогічний опис lineptr в прикладі з сортуванням. Ініціалізатор це просто список символьних рядків; кожен рядок присвоюється відповідної позиції в масиві. Більш точно, символи i -ої рядки поміщаються в якесь інше місце, а її покажчик зберігається в name [i]. Оскільки розмір масиву name не вказано, компілятор сам підраховує кількість ініціалізаторів і відповідно встановлює правильне число.

#### 5.10. Покажчики та багатовимірні масиви

Початківці вивчати мову "с" іноді стають в безвихідь перед питанням про відмінності між двовимірним масивом і масивом покажчиків, таким як name у наведеному вище прикладі. Якщо є описи

int a[10][10]; int \*b[10];

то a і b можна використовувати подібним чином в тому сенсі, що як a [5] [5], так і b [5] [5] є законними посиланнями на окреме число типу int. Але a - справжній масив: під нього відводиться 100 осередків пам'яті і для знаходження будь-якого зазначеного елемента проводяться звичайні обчислення з прямокутними індексами. Для b, однак, опис виділяє тільки 10 вказівників; кожен покажчик повинен бути встановлений так, щоб він вказував на масив цілих. Якщо припустити, що кожен з них вказує на масив з 10 елементів, то тоді десь буде відведено 100 осередків пам'яті плюс ще десять осередків для покажчиків.

Таким чином, масив покажчиків використовує дещо більший обсяг пам'яті і може вимагати наявність явного кроку ініціалізації. Але при цьому виникають дві переваги: доступ до елементу здійснюється побічно через покажчик, а не за допомогою множення і складання, і рядки масиву можуть мати різні довжини. Це означає, що кожен елемент b не повинен обов'язково вказувати на вектор з 10 елементів; деякі можуть вказувати на вектор з двох елементів, інші - з двадцяти, а треті можуть взагалі ні на що не вказувати.

Хоча ми вели це обговорення в термінах цілих, без сумніву, найчастіше масиви покажчиків використовуються так, як ми продемонстрували на функції month\_name, - для зберігання символьних рядків різної довжини.

**Вправа 5-6**

Перепишіть функції day\_of\_year і month\_day, використовуючи замість індексації покажчики.

#### 5.11. Командний рядок аргументів

Системні засоби, на які спирається реалізація мови "с", дозволяють передавати командний рядок аргументів або параметрів починаючої виконуватися програмою. Коли функція main викликається до виконання, вона викликається з двома аргументами. Перший аргумент (умовно званий argc) вказує число аргументів в командному рядку, з якими відбувається звернення до програми; другий аргумент (argv) є покажчиком на масив символьних рядків, що містять ці аргументи, по одному в рядку. Робота з такими рядками - це звичайне використання багаторівневих покажчиків.

Найпростішу ілюстрацію цієї можливості і необхідних при цьому описів дає програма echo, яка просто друкує в один рядок аргументи командного рядка, розділяючи їх пробілами. Таким чином, якщо дана команда

echo hello, world

то виходом буде

hello, world

за угодою argv [0] є ім'ям, по яким викликається програма, так що argc щонайменше дорівнює 1. У наведеному вище прикладі argc дорівнює 3, а argv [0], argv [1] і argv [2] рівні відповідно "echo "," hello, "і" world ". Першим фактичним аргументом є argv [1], а останнім - argv [argc-1]. Якщо argc дорівнює 1, то за ім'ям програми не слід ніякої командного рядка аргументів. Все це показано в echo:

main(argc, argv) /\* echo arguments; 1st version \*/

int argc;

char \*argv[];

{

int i;

for (i = 1; i < argc; i++)

printf("%s%c", argv[i], (i<argc-1) ? ' ' : '\n');

}

Оскільки argv є покажчиком на масив покажчиків, то існує кілька способів написання цієї програми, які використовують працю з покажчиком, а не з індексацією масиву. Ми продемонструємо два варіанта.

main(argc, argv) /\* echo arguments; 2nd version \*/

int argc;

char \*argv[];

{

while (--argc > 0)

printf("%s%c",\*++argv, (argc > 1) ? ' ' : '\n');

}

Так як argv є покажчиком на початок масиву строк- аргументів, то, збільшивши його на 1 (++ argv), ми змушуємо його вказувати на справжній аргумент argv [1], а не на argv [0]. Кожне наступне збільшення пересуває його на наступний аргумент; при цьому \* argv стає дороговказом на цей аргумент. Одночасно величина argc зменшується; коли вона звернеться в нуль, всі аргументи будуть уже надруковані.

Інший варіант:

main(argc, argv) /\* echo arguments; 3rd version \*/

int argc;

char \*argv[];

{

while (--argc > 0)

printf((argc > 1) ? "%s" : "%s\n", \*++argv);

}

Ця версія показує, що аргумент формату функції printf може бути виразом, точно так же, як і будь-який інший. Таке використання зустрічається не дуже часто, але його все ж варто запам'ятати.

Як другий приклад, давайте внесемо деякі удосконалення в програму відшукання заданої комбінації символів з "лекції №4" . Якщо ви пам'ятаєте, ми помістили шукану комбінацію глибоко всередину програми, що очевидно є абсолютно незадовільним. Дотримуючись утиліті grep системи UNIX, давайте змінимо програму так, щоб ця комбінація вказувалася як перший аргумент рядка.

#define maxline 1000

main(argc, argv) /\* find pattern from first argument \*/

int argc;

char \*argv[];

{

char line[maxline];

if (argc != 2)

printf ("usage: find pattern\n");

else

while (getline(line, maxline) > 0)

if (index(line, argv[1] >= 0)

printf("%s", line);

}

Тепер може бути розвинена основна модель, що ілюструє подальше використання покажчиків. Припустимо, що нам треба передбачити два необов'язкових аргументи. Один стверджує: "надрукувати всі рядки за винятком тих, які містять цю комбінацію", другий говорить: "перед кожною виведеної рядком повинен друкуватися її номер".

Загальноприйнятим угодою в "с" - програмах є те, що аргумент, що починається зі знака мінус, вводить необов'язковий ознака або параметр. Якщо ми, для того, щоб повідомити про інверсії, виберемо -x, а для вказівки про нумерацію потрібних рядків виберемо -n ("номер"), то команда

find -x -n the

при введені данних

now is the time

for all good men

to come to the aid

of their party.

Повинні отримати

2:for all good men

Потрібно, щоб необов'язкові аргументи могли розташовуватися в довільному порядку, і щоб інша частина програми не залежала від кількості фактично присутніх аргументів. Зокрема, виклик функції index не повинен містити посилання на argv [2], коли присутній один необов'язковий аргумент, і на argv [1], коли його немає. Більш того, для користувачів зручно, щоб необов'язкові аргументи можна було об'єднати в вигляді:

find -nx the

ось сама программа:

#define maxline 1000

main(argc, argv) /\* find pattern from first argument \*/

int argc;

char \*argv[];

{

char line[maxline], \*s;

long lineno = 0;

int except = 0, number = 0;

while (--argc > 0 && (\*++argv)[0] == '-')

for (s = argv[0]+1; \*s != '\0'; s++)

switch (\*s) {

case 'x':

except = 1;

break;

case 'n':

number = 1;

break;

default:

printf("find: illegal option %c\n", \*s);

argc = 0;

break;

}

if (argc != 1)

printf("usage: find -x -n pattern\n");

else

while (getlinе(line, maxline) > 0) {

lineno++;

if ((index(line, \*argv) >= 0) != except) \

if (number)

printf("%ld: ", lineno);

printf("%s", line);

}

}

}

аргумент argv збільшується перед кожним необов'язковим аргументом, в той час як аргумент argc зменшується. Якщо немає помилок, то в кінці циклу величина argc повинна дорівнювати 1, а \* argv має вказувати на задану комбінацію. Зверніть увагу на те, що \* ++ argv є покажчиком аргументної рядки; (\* ++ argv) [0] - її перший символ. Круглі дужки тут потрібні, тому що без них вираз б прийняло абсолютно відмінний (і неправильний) вид \* ++ (argv [0]). Інший правильною формою була б \*\*++argv.

**Вправа 5-7**

Напишите программу add, вычисляющую обратное польское выражение из командной строки. Например,

add 2 3 4 + \*

вычисляет 2\*(3+4).

**Вправа 5-8**

Модифікуйте програми entab і detab (зазначені в якості вправ в "лекції №1") так, щоб вони отримували список табуляціонних зупинок в якості аргументів. Якщо аргументи відсутні, використовуйте стандартну установку табуляцій.

**Вправа 5-9**

Розширте entab і detab таким чином, щоб вони сприймали скорочену нотацію

entab m +n

яка б означала табуляціонні зупинки через кожні n стовпців, починаючи зі стовпчика m. Виберіть зручне (для користувача) поведінка функції за замовчуванням.

**Вправа 5-10**

Напишіть програму для функції tail, що друкує останні n рядків зі свого файлу введення. Нехай за замовчуванням n дорівнює 10, але це число може бути змінено за допомогою необов'язкового аргументу, так що

tail -n

друкує останні n рядків. Програма повинна діяти раціонально, якими б нерозумними не були б введення або значення n. Складіть програму так, щоб вона оптимальним чином використовувала доступну пам'ять: рядки повинні зберігатися, як у функції sort, а не в двовимірному масиві фіксованого розміру.

#### 5.12. Покажчики на функції

У мові "с" самі функції не є змінними, але є можливість визначити покажчик на функцію, який можна обробляти, передавати іншим функціям, поміщати в масиви і т.д. Ми проілюструємо це, провівши модифікацію написаної раніше програми сортування так, щоб при завданні необов'язкового аргументу -n вона б сортувала рядки введення чисельно, а не лексикографічно.

Сортування часто складається з трьох частин - порівняння, яке визначає упорядкування будь-якої пари об'єктів, перестановки, що змінює їх порядок, і алгоритму сортування, що здійснює порівняння і перестановки до тих пір, поки об'єкти не розташуються в потрібному порядку. Алгоритм сортування не залежить від операцій порівняння і перестановки, так що, передаючи в нього різні функції порівняння і перестановки, ми можемо організувати сортування за різними критеріями. Саме такий підхід використовується в нашій новій програмі сортування.

Як і раніше, лексикографічні порівняння двох рядків здійснюється функцією strcmp, а перестановка функцією swap; нам потрібна ще функція numcmp, що порівнює два рядки на основі чисельного значення і повертає умовне вказівку того ж виду, що й strcmp. Ці три функції описуються в main і покажчики на них передаються в sort. У свою чергу функція sort звертається до цих функцій через їх покажчики. Ми урізали обробку помилок в аргументах з тим, щоб зосередитися на головних питаннях.

#define lines 100 /\* max number of lines

to be sorted \*/

main(argc, argv) /\* sort input lines \*/

int argc;

char \*argv[];

{

char \*lineptr[lines]; /\* pointers to text lines \*/

int nlines; /\* number of input lines read \*/

int strcmp(), numcmp(); /\* comparsion functions \*/

int swap(); /\* exchange function \*/

int numeric = 0; /\* 1 if numeric sort \*/

if(argc>1 && argv[1][0] == '-' && argv[1][1]=='n')

numeric = 1;

if(nlines = readlines(lineptr, lines)) >= 0) {

if (numeric)

sort(lineptr, nlines, numcmp, swap);

else

sort(lineptr, nlines, strcmp, swap);

writelines(lineptr, nlines);

} else

printf("input too big to sort\n");

}

Тут strcmp, numcmp і swap - адреси функцій; так як відомо, що це функції, операція & тут не потрібна абсолютно аналогічно тому, як вона не потрібна і перед ім'ям масиву. Передача адрес функцій організовується компілятором.

Другий крок полягає в модифікації sort:

sort(v, n, comp, exch) /\* sort strings v[0] ... v[n-1] \*/

char \*v[]; /\* into increasing order \*/

int n;

int (\*comp)(), (\*exch)();

{

int gap, i, j;

for(gap = n/2; gap > 0; gap /= 2)

for(i = gap; i < n; i++)

for(j = i-gap; j >= 0; j -= gap) {

if((\*comp)(v[j], v[j+gap]) <= 0)

break;

(\*exch)(&v[j], &v[j+gap]);

}

}

Тут слід звернути певну увагу на описи. Опис

int (\*comp)()

каже, что comp є покажчики на функцію, яка Повертає значення типу int. Перші круглі дужки тут потрібні; без них опис

int \*comp()

говорило б, що comp є функцією, що повертає покажчик на цілі, що, звичайно, зовсім інша річ.

Використання comp в рядку

if ((\*comp)(v[j], v[j+gap]) <= 0)

повністю узгоджується з описом: comp - покажчик на функцію, \* comp - сама функція, а

(\*comp)(v[j], v[j+gap])

- звернення до неї. Круглі дужки необхідні для правильного об'єднання компонентів.

Ми вже наводили функцію strcmp, що порівнює два рядки по першому чисельним значенням:

numcmp(s1, s2) /\* compare s1 and s2 numerically \*/

char \*s1, \*s2;

{

double atof(), v1, v2;

v1 = atof(s1);

v2 = atof(s2);

if(v1 < v2)

return(-1);

else if(v1 > v2)

return(1);

else

return (0);

}

Заключний крок полягає в додаванні функції swap, переставляти два покажчика. Це легко зробити, безпосередньо використовуючи те, що ми виклали раніше в цій лекції.

swap(px, py) /\* interchange \*px and \*py \*/

char \*px[], \*py[];

{

char \*temp;

temp = \*px;

\*px = \*py;

\*py = temp;

}

Є безліч інших необов'язкових аргументів, які можуть бути включені в програму сортування: деякі з них складають цікаві вправи.

**Вправа 5-11**

Модифікуйте sort таким чином, щоб вона працювала з міткою -r, що вказує на сортування в зворотному (спадання) порядку. Звичайно, -r повинна працювати с -n.

**Вправа 5-12**

Додайте необов'язковий аргумент -f, який об'єднує разом великі та малі літери, так щоб відмінність регістрів не враховувався під час сортування: дані з верхнього і нижнього регістрів упорядковано разом, так що буква 'а' прописне і 'а' рядкове виявляються сусідніми, а не розділеними цілим алфавітом.

**Вправа 5-13**

Додайте необов'язковий аргумент -d ("словникове упорядкування"), при наявності якого порівнюються тільки букви, числа і прогалини. Подбайте про те, щоб ця функція працювала і разом с -f.

**Вправа 5-14**

Додайте можливість обробки полів, так щоб можна було сортувати поля всередині рядків. Кожне поле має сортуватися відповідно до незалежним набором необов'язкових аргументів. (Предметний покажчик цієї книги сортувалося за допомогою аргументів -df для категорії покажчика і з -n для номерів сторінок).