### 4. Функція і структури програм

Функції розбивають великі обчислювальні завдання на маленькі підзадачі і дозволяють використовувати в роботі те, що вже зроблено іншими, а не починати кожен раз з пустого місця. Відповідні функції часто можуть приховувати в собі деталі проведених в різних частинах програми операцій, знати які немає необхідності, проясняючи тим самим всю програму, як ціле, і полегшуючи муки при внесенні змін.

Мова "C" розроблявся з прагненням зробити функції ефективними та зручними для використання; "C" - програми зазвичай складаються з великого числа маленьких функцій, а не з декількох великих. Програма може розміщуватися в одному або декількох вихідних файлах будь-яким зручним чином; вихідні файли можуть компілюватися окремо і завантажуватися разом поряд зі скомпільованими раніше функціями з бібліотек. Ми тут не будемо вдаватися в деталі цього процесу, оскільки вони залежать від використовуваної системи.

Більшість програмістів добре знайомі з "бібліотечними" функціями для введення і виведення / getchar, putchar / і для чисельних розрахунків / sin, cos, sqrt /. У цій лекції ми повідомимо більше про написання нових функцій.

#### 4.1. Основні відомості

Для початку давайте розробимо і складемо програму друку кожного рядка введення, яка містить певну комбінацію символів. / Це - спеціальний випадок утиліти grep системи "UNIX" /. Наприклад, при пошуку комбінації "the" в наборі рядків

now is the time

for all good

men to come to the aid

of their party

в якості виходу отримаємо

now is the time

men to come to the aid

of their party

основна схема виконання завдання чітко розділяється на три частини:

while (є ще рядок)

if (рядок містить потрібну комбінацію)

вивід цього рядка

Звичайно, можливо запрограмувати всі дії у вигляді однієї основної процедури, але краще використовувати природну структуру завдання і представити кожну частину у вигляді окремої функції. З трьома маленькими шматками легше мати справу, ніж з одним великим, тому що окремі не відносяться до суті справи деталі можна включити в функції і зменшити можливість небажаних взаємодій. Крім того, ці шматки можуть виявитися корисними самі по собі.

"Поки є ще рядок" - це getline, функція, яку ми запрограмували в "лекції №1", а "висновок цього рядка" - це функція printf, яку вже хтось підготував для нас. Це означає, що нам залишилося тільки написати процедуру для визначення, чи містить рядок дану комбінацію символів чи ні. Ми можемо вирішити цю проблему, запозичивши розробку з PL / 1: функція index (s, t) повертає позицію, або індекс, рядки s, де починається рядок t, і -1, якщо s не містить t. В якості початкової позиції ми використовуємо 0, а не 1, тому що в мові "C" масиви починаються з позиції нуль. Коли нам надалі знадобиться перевіряти на збіг більш складні конструкції, нам доведеться замінити тільки функцію index; інша частина програми залишиться тією ж самою.

Після того, як ми витратили стільки зусиль на розробку, написання програми в деталях не становить труднощів. Нижче наводиться цілком вся програма, так що ви можете бачити, як з'єднуються один з одним окремі частини. Комбінація символів, по якій проводиться пошук, виступає поки в якості символьного рядка в аргументі функції index, що не є самим загальним механізмом. Ми скоро повернемося до обговорення питання про ініціалізації символьних масивів і в "лекції №5" покажемо, як зробити комбінацію символів параметром, яким присвоюється значення в ході виконання програми. Програма також містить новий варіант функції getline; вам може виявитися корисним порівняти його з варіантом з "лекції №1".

#define maxline 1000

main() /\* find all lines matching a pattern \*/

{

char line[maxline];

while (getline(line, maxline) > 0)

if (index(line, "the") >= 0)

printf("%s", line);

}

getline(s, lim) /\* get line into s, return length \*

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);

}

index(s,t) /\* return index of t in s,-1 if none \*/

char s[], 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 (t[k] == '\0')

return(i);

}

return(-1);

}

Кожна функція має вигляд ім'я (список аргументів, якщо вони є) опису аргументів, якщо вони є

{

Описи та оператори, якщо вони є

}

Як і вказується, деякі частини можуть бути відсутні; мінімальної функцією є

dummy () { }

яка не робить ніяких дій.

/ Така нічого не робляча функція іноді виявляється зручною для збереження місця для подальшого розвитку програми /. Якщо функція повертає щось відмінне від цілого значення, то перед її ім'ям може стояти покажчик типу; це питання обговорюється в наступному розділі.

Програмою є просто набір визначень окремих функцій. Зв'язок між функціями здійснюється через аргументи і повертаються функціями значення / в цьому випадку /; її можна також здійснювати через зовнішні змінні. функції можуть розташовуватися в вихідному файлі в будь-якому порядку, а сама вихідна програма може розміщуватися на декількох файлах, але так, щоб жодна функція не розщеплюється.

Оператор return служить механізмом для повернення значення з викликаної функції в функцію, яка до неї звернулася. За return може слідувати будь-який вираз:

return (вираз)

Викликаюча функція може ігнорувати значення, що повертається, якщо вона цього забажає. Більш того, після return може не бути взагалі ніякого виразу; в цьому випадку в викликаючу програму не передається ніякого значення. Управління також повертається в зухвалу програму без передачі будь-якого значення і в тому випадку, коли при виконанні ми "провалюємось" на кінець функції, досягаючи закривається правої фігурної дужки. Eсли функція повертає значення з одного місця і не повертає ніякого значення з іншого місця, це не є незаконним, але може бути ознакою якихось неприємностей. У будь-якому випадку "значенням" функції, яка не повертає значення, безсумнівно буде сміття. Отладочная програма lint перевіряє такі помилки.

Механіка компіляції та завантаження "C" - програм, розташованих в декількох вихідних файлах, змінюється від системи до системи. В системі "UNIX", наприклад, цю роботу виконує команда 'cc', згадана в "лекції №1". Припустимо, що три функції знаходяться в трьох різних файлах з іменами main.с, getline.c і index.с. тоді команда

cc main.c getline.c index.c

компілює ці три файли, поміщає отриманий настроюється об'єктний код у файли main.o, getline.o і index.o і завантажує їх всіх в виконуваний файл під назвою a.out.

Якщо є якась помилка, скажімо в main.c, то цей файл можна перекомпілювати окремо і завантажити разом з попередніми об'єктними файлами по команді

cc main.c getline.o index.o

Команда 'cc' використовує угоду про найменування з ".с" і ".про" для того, щоб відрізнити вихідні файли від об'єктних.

**Вправа 4-1**

Складіть програму для функції rindex (s, t), яка повертає позицію самого правого входження t в s і -1, якщо s не містить t.

#### 4.2. Функції, які повертають нецілі значення

До сих пір жодна з наших програм не містила будь-якого опису типу функції. Справа в тому, що за замовчуванням функція неявно описується своєю появою в вираженні або операторі, як, наприклад, в

while (getline(line, maxline) > 0)

Якщо деякий ім'я, яке не було описано раніше, з'являється в вираженні і за ним слід ліва кругла дужка, то воно по контексту вважається ім'ям деякої функції. Крім того, за замовчуванням передбачається, що ця функція повертає значення типу int. Так як в виразах char перетворюється в int, то немає необхідності описувати функції, які повертають char. Ці припущення покривають більшість випадків, включаючи всі наведені досі приклади.

Але що відбувається, якщо функція повинна повернути значення якогось іншого типу? Багато чисельні функції, такі як sqrt, sin і cos повертають double; інші спеціальні функції повертають значення інших типів. Щоб показати, як чинити в цьому випадку, давайте напишемо і використовуємо функцію atof (s), яка перетворює рядок s в еквівалентне їй плаваюче число подвійної точності. Функція atof є розширенням atoi, варіанти якої ми написали в "лекції №2" і "№3"; вона обробляє необов'язково знак і десяткову точку, а також цілу і дробову частину, кожна з яких може як бути присутнім, так і бути відсутнім. / Ця процедура перетворення введення не дуже високої якості; інакше вона б зайняла більше місця, ніж нам хотілося б /.

По-перше, сама atof повинна описувати тип поверненого нею значення, оскільки він відрізняється від int. Так як в виразах тип float перетворюється в double, то немає ніякого сенсу в тому, щоб atof повертала float; ми можемо з рівним успіхом скористатися додатковою точністю, так що ми вважаємо, що повертається значення типу double. Ім'я типу має стояти перед ім'ям функції, як показується нижче:

double atof(s) /\* convert string s to double \*/

char s[];

{

double val, power;

int i, sign;

for(i=0; s[i]==' ' || s[i]=='\n' || s[i]=='\t'; i++)

; /\* skip white space \*/

sign = 1;

if (s[i] == '+' || s[i] == '-') /\* sign \*/

sign = (s[i++] == '+') ? 1 : -1;

for (val = 0; s[i] >= '0' && s[i] <= '9'; i++)

val = 10 \* val + s[i] - '0';

if (s[i] == '.')

i++;

for (power = 1; s[i] >= '0' && s[i] <= '9'; i++) {

val = 10 \* val + s[i] - '0';

power \*= 10;

}

return(sign \* val / power);

}

Другим, але настільки ж важливим, є те, що викликаюча функція повинна оголосити про те, що atof повертає значення, відмінне від int типу. Таке оголошення демонструється на прикладі наступного примітивного настільного калькулятора / ледь придатного для підведення балансу в чековій книжці /, який зчитує по одному числу на рядок, причому це число може мати знак, і складає все числа, друкуючи суму після кожного введення.

#define maxline 100

main() /\* rudimentary desk calkulator \*/

{

double sum, atof();

char line[maxline];

sum = 0;

while (getline(line, maxline) > 0)

printf("\t%.2f\n",sum+=atof(line));

}

*опим*

double sum, atof();

каже, що sum є змінною типу double, і що atof є функцією, що повертає значення типу double. Ця мнемоніка означає, що значеннями як sum, так і atof (...) є плаваючі числа подвійної точності.

Якщо функція atof НЕ буде описана явно в обох місцях, то в "C" передбачається, що вона повертає ціле значення, і ви отримаєте безглузду відповідь. Якщо сама atof і звернення до неї в main мають несумісні типи і знаходяться в одному і тому ж файлі, то це буде виявлено компілятором. Але якщо atof була скомпільована окремо / що більш ймовірно /, то ця невідповідність НЕ буде зафіксована, так що atof буде повертати значення типу double, з яким main буде звертатися, як з int, що призведе до безглуздих результатів. / Програма lint виловлює цю помилку/.

Маючи atof, ми, в принципі, могли б з її допомогою написати atoi (перетворення рядка в int ):

atoi(s) /\* convert string s to integer \*/

char s[];

{

double atof();

return(atof(s));

}

Зверніть увагу на структуру описів і оператор return. Значення виразу в

return (вираз)

завжди перетворюється до типу функції перед виконанням самого повернення. Тому при появі в операторі return значення функції atof, що має тип double, автоматично перетворюється в int, оскільки функція atoi повертає int. (Як обговорювалося в "лекції №2", перетворення значення з плаваючою точкою до типу int здійснюється за допомогою відкидання дробової частини).

**Вправа 4-2**

Розширте atof таким чином, щоб вона могла працювати з числами такого виду

123.45е-6

де за числом з плаваючою крапкою може слідувати 'e' і показник експоненти, можливо зі знаком.

#### 4.3. Ще про аргументи функцій

В "лекції №1" ми вже обговорювали той факт, що аргументи функцій передаються за значенням, тобто викликана функція отримує свою тимчасову копію кожного аргументу, а не його адреса. Це означає, що викликана функція не може впливати на вихідний аргумент на функції що викликає. Усередині функції кожен аргумент по суті є локальною змінною, яка ініціалізується тим значенням, з яким до цієї функції звернулися.

Якщо в якості аргументу функції виступає ім'я масиву, то передається адреса початку цього масиву; самі елементи не копіюються. Функції зможете змінити елементи масиву, використовуючи індексацію і адреса початку. Таким чином, масив передається по посиланню. В "лекції №5" ми обговоримо, як використання покажчиків дозволяє функцій впливати на відмінні від масивів змінні в викликаючих функціях.

Між іншим, не існує повністю задовільного способу написання функції яку переносять зі змінним числом аргументів. Справа в тому, що немає переносимого способу, за допомогою якого викликана функція могла б визначити, скільки аргументів було фактично передано їй в даному зверненні. Таким чином, ви, наприклад, не можете написати дійсно функцію що переноситься, яка буде обчислювати максимум від довільного числа аргументів, як роблять вбудовані функції max в фортране і PL / 1.

Зазвичай з випадком змінного числа аргументів безпечно мати справу, якщо викликана функція не використовує аргументів, які їй насправді не були передані, і якщо типи узгоджуються. Найпоширеніша в мові "C" функція зі змінним числом - printf. Вона отримує з першого аргументу інформацію, що дозволяє визначити кількість інших аргументів і їх типи. функція printf працює абсолютно неправильно, якщо викликає функція передає їй недостатня кількість аргументів, або якщо їх типи не узгоджуються з типами, зазначеними в першому аргументі. Ця функція не є переносною і має модифікуватися при використанні в різних умовах.

Якщо ж типи аргументів відомі, то кінець списку аргументів можна відзначити, використовуючи якусь угоду; наприклад, вважаючи, що деяке спеціальне значення аргументу (часто нуль) є ознакою кінця аргументів.

#### 4.4. Зовнішні змінні

Програма на мові "C" складається з набору зовнішніх об'єктів, які є або змінними, або функціями. Термін "зовнішній" використовується головним чином в протиставлення терміну "внутрішній", яким описуються аргументи і автоматичні змінні, певні всередині функцій. Зовнішні змінні визначені поза будь-якої функції і, таким чином, потенційно доступні для багатьох функцій. Самі функції завжди є зовнішніми, тому що правила мови "C" не дозволяють визначати одні функції всередині інших. За замовчуванням зовнішні змінні є також і "глобальними", так що всі посилання на таку змінну, що використовують одине і те ж ім'я (навіть з функцій, скомпільованих незалежно), будуть посиланнями на одне і те ж. У цьому сенсі зовнішні змінні аналогічні змінним common в фортране і external в PL / 1. Пізніше ми покажемо, як визначити зовнішні змінні і функції таким чином, щоб вони були доступні не глобально, а тільки в межах одного вихідного файлу.

В силу своєї глобальної доступності зовнішні змінні надають іншу, відмінну від аргументів і значень, можливість для обміну даними між функціями. Якщо ім'я зовнішньої змінної будь-яким чином описано, то будь-яка функція має доступ до цієї змінної, посилаючись до неї з цього імені.

У випадках, коли зв'язок між функціями здійснюється за допомогою великого числа змінних, зовнішні змінні виявляються більш зручними і ефективними, ніж використання довгих списків аргументів. Як, проте, зазначалося в "лекції №1", це міркування слід використовувати з певною обережністю, так як воно може погано відбитися на структурі програм і приводити до програм з великим числом зв'язків за даними між функціями.

Друга причина використання зовнішніх змінних пов'язана з ініціалізацією. Зокрема, зовнішні масиви можуть бути ініційовані, а автоматичні ні. Ми розглянемо питання про ініціалізації в кінці цієї лекції.

Третя причина використання зовнішніх змінних обумовлена ​​їх областю дії і часом існування. Автоматичні змінні є внутрішніми по відношенню до функцій; вони виникають при вході в функцію і зникають при виході з неї. Зовнішні змінні, навпаки, існують постійно. Вони не з'являються і не зникають, так що можуть зберігати свої значення в період від одного звернення до функції до іншого. В силу цього, якщо дві функції використовують деякі загальні дані, причому жодна з них не звертається до іншого, то часто найбільш зручним виявляється зберігати ці загальні дані у вигляді зовнішніх змінних, а не передавати їх в функцію і назад за допомогою аргументів.

Давайте продовжимо обговорення цього питання на великому прикладі. Завдання полягатиме в написанні іншої програми для калькулятора, кращої, ніж попередня. Тут допускаються операції +, -, \*, / і знак = (для видачі відповіді). Замість інфіксного уявлення калькулятор буде використовувати зворотну польську нотацію, оскільки її дещо легше реалізувати. У зворотної польської нотації знак слідує за операндами; інфіксний вираз типу

(1-2)\*(4+5)=

Записується у вигляді

12-45+\*=

*Круглі* дужки при цьому не потрібні

Реалізація виявляється вельми простою. Кожен операнд міститься в стеках; коли надходить знак операції, потрібне число операндов (два для бінарних операцій) виймається, до них застосовується операція і результат прямує назад в стек. Так в наведеному вище прикладі 1 і 2 поміщаються в стек і потім замінюються їх різницею, -1. Після цього 4 і 5 вводяться в стек і потім замінюються своєю сумою, 9. Далі числа -1 і 9 замінюються в стеці на їх твір, рівний -9. Операція = друкує верхній елемент стека, не видаляючи його (так що проміжні обчислення можуть бути перевірені).

Самі операції приміщення чисел в стек і їх вилучення дуже прості, але, у зв'язку з включенням в справжню програму виявлення помилок і відновлення, вони виявляються досить довгими. Тому краще оформити їх у вигляді окремих функцій, ніж повторювати відповідний текст всюди в програмі. Крім того, потрібна окрема функція для вибірки з введення наступної операції або операнда. Таким чином, структура програми має вигляд:

while( надходить операція або операнд, а не кінець)

if ( число )

помістити його в стек

еlse if ( операція )

вийняти операнди зі стека  
         виконати операцію  
         помістити результат в стек

else

помилка

Основне питання, яке ще не було обговорено, полягає в тому, де помістити стек, тобто які процедури зможуть звертатися до нього безпосередньо. Одна з таких можливостей полягає в приміщенні стека в main і передачі самого стека і поточної позиції в стеці функцій, які працюють зі стеком. Але функції main немає необхідності мати справу зі змінними, які керують стеком; їй природно міркувати в термінах приміщення чисел в стек і вилучення їх звідти. В силу цього ми вирішили зробити стек і пов'язану з ним інформацію зовнішніми змінними, доступними функціями push (приміщення в стек) і pop (витяг з стека), але не main.

Переклад цієї схеми в програму досить простий. Провідна програма є по суті великим перемикачем за типом операції або операнду; це, мабуть, більш характерне застосування перемикача, ніж те, яке було продемонстровано в "лекції №3" .

#define maxop 20 /\* max size of operand, operator \*/

#define number '0' /\* signal that number found \*/

#define toobig '9' /\* signal that string is too big \*/

main() /\* reverse polish desk calculator \*/

{

int tupe;

char s[maxop];

double op2,atof(),pop(),push();

while ((tupe=getop(s,maxop)) !=EOF);

switch(tupe) {

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("zero divisor popped\n");

break;

case '=':

printf("\t%f\n",push(pop()));

break;

case 'c':

clear();

break;

case toobig:

printf("%.20s ... is too long\n",s);

break;

}

}

#define maxval 100 /\* maximum depth of val stack \*/

int sp = 0; /\* stack pointer \*/

double val[maxval]; /\*value stack \*/

double push(f) /\* push f onto value stack \*/

double f;

{

if (sp < maxval)

return(val[sp++] =f);

else {

printf("error: stack full\n");

clear();

return(0);

}

}

double pop() /\* pop top value from steack \*/

{

if (sp > 0)

return(val[--sp]);

else {

printf("error: stack empty\n");

clear();

return(0);

}

}

clear() /\* clear stack \*/

{

sp=0;

}

Команда C очищає стек за допомогою функції clear, яка також використовується в разі помилки функціями push і pop. До функції getop ми дуже скоро повернемося.

Як вже говорилося в "лекції №1", змінна є зовнішньою, якщо вона визначена поза тілом якої б то не було функції. Тому стек і покажчик стека, які повинні використовуватися функціями push, pop і clear, визначені поза цих трьох функцій. Але сама функція main не посилається ні до стека, ні до покажчика стека - їх участь ретельно замасковано. В силу цього частина програми, відповідна операції =, використовує конструкцію

push(pop())

для того, щоб проаналізувати верхній елемент стека, не змінюючи його.

Отметим также, что так как операции + и \* коммутативны, порядок, в котором объединяются извлеченные операнды, несущественен, но в случае операций - и / необходимо различать левый и правый операнды.

**Вправа 4-3**

Наведена основна схема допускає безпосереднє розширення можливостей калькулятора. Увімкніть операцію ділення по модулю /% / і унарний мінус. Увімкніть команду "стерти", яка видаляє верхній елемент стека. Введіть команди для роботи зі змінними. / Це просто, якщо імена змінних будуть складатися з однієї літери з наявних двадцяти шести букв/.

4.5. Правила, що визначають область дії

Функції і зовнішні змінні, що входять до складу "C" - програми, не зобов'язані компілюватиметься одночасно; програма мовою оригіналу може розташовуватися в декількох файлах, і раніше скомпільовані процедури можуть завантажуватися з бібліотек. Два питання становлять інтерес:

1. Як слід складати описи, щоб змінні правильно сприймалися під час компіляції?
2. Як слід складати описи, щоб забезпечити правильний зв'язок частин програми при завантаженні?

4.5.1. Область дії

Областю дії імені є та частина програми, в якій це ім'я визначено. Для автоматичної змінної, описаної на початку функції, областю дії є та функція, в якій описано ім'я цієї змінної, а змінні з різних функцій, що мають однакове ім'я, вважаються не відносяться один до одного. Це ж справедливо і для аргументів функцій.

Область дії зовнішньої змінної простягається від точки, в якій вона оголошена в вихідному файлі, до кінця цього файлу. Наприклад, якщо val, sp, push, pop і clear визначені в одному файлі в порядку, зазначеному вище, а саме:

int sp = 0;

double val[maxval];

double push(f) {...}

double pop() {...}

clear() {...}

то змінні val і sp можна використовувати в push, pop і clear прямо по імені; ніякі додаткові описи не потрібні.

З іншого боку, якщо потрібно послатися на зовнішню змінну до її визначення, або якщо така змінна визначена в файлі, відмінному від того, в якому вона використовується, то необхідно опис extern.

Важливо розрізняти опис зовнішньої змінної і її визначення. опис вказує властивості змінної / її тип, розмір і т.д. /; визначення ж викликає ще і відведення пам'яті. Якщо поза якої б то не було функції з'являються рядки

int sp;

double val[maxval];

то вони визначають зовнішні змінні sp і val, викликають відведення пам'яті для них і служать в якості опису для іншої частини цього вихідного файлу. У той же час рядки

extern int sp;

extern double val[];

описують в решті частини цього вихідного файлу змінну sp як int, а val як масив типу double / розмір якого вказано в іншому місці /, але не створюють змінних і не відводять їм місця в пам'яті.

У всіх файлах, що становлять вихідну програму, має міститися лише одне визначення зовнішньої змінної; інші файли можуть містити описи extern для доступу до неї. / Опис extern може бути і в тому файлі, де знаходиться визначення /. Будь-яка ініціалізація зовнішньої змінної проводиться тільки у визначенні. В ухвалі повинні вказуватися розміри масивів, а в описі extern цього можна не робити.

Хоча подібна організація наведеної вище програми і малоймовірна, але val і sp могли б бути визначені і ініціалізовані в одному файлі, а функція push, pop і clear визначені в іншому. В цьому випадку для зв'язку були б необхідні наступні визначення і опису:

в файлі 1:

----------

int sp = 0; /\* stack pointer \*/

double val[maxval]; /\* value stack \*/

в файлі 2:

----------

extern int sp;

extern double val[];

double push(f) {...}

double pop() {...}

clear() {...}

так як опису extern "у файлі 1 'знаходяться вище і поза трьох зазначених функцій, вони відносяться до всіх ним; одного набору описів досить для всього 'файлу 2'.  
Для програм великого розміру обговорювана пізніше в цій лекції можливість включення файлів, #include, дозволяє мати в усій програмі тільки одну копію описів extern і вставляти її в кожен вихідний файл під час його компіляції.

Звернемося тепер до функції getop, яка обирає з файлу введення наступну операцію або операнд. Основна задача проста: пропустити прогалини, знаки табуляції і нові рядки. Якщо наступний символ відмінний від цифри і десяткового дробу, то повернути його. В іншому випадку зібрати рядок цифр / вона може включати десяткову точку / і повернути number як сигнал про те, що вибрано число.

процедура істотно ускладнюється, якщо прагнути правильно обробляти ситуацію, коли введене число виявляється занадто довгим. Функція getop зчитує цифри поспіль / можливо з десятковою крапкою / і запам'ятовує їх, поки послідовність не переривається. Якщо при цьому не відбувається переповнення, то функція повертає number і рядок цифр. Якщо ж число виявляється занадто довгим, то getop відкидає іншу частину рядка з файлу введення, так що користувач може просто передрукувати цей рядок з місця помилки; функція повертає toobig як сигнал про переповнення.

getop(s, lim) /\* get next operator or operand \*/

char s[];

int lim;

{

int i, c;

while((c=getch())==' '|| c=='\t' || c=='\n')

;

if (c != '.' && (c < '0' || c > '9'))

return(c);

s[0] = c;

for(i=1; (c=getchar()) >='0' && c <= '9'; i++)

if (i < lim)

s[i] = c;

if (c == '.') { /\* collect fraction \*/

if (i < lim)

s[i] = c;

for(i++;(c=getchar()) >='0' && c<='9';i++)

if (i < lim)

s[i] =c;

}

if (i < lim) { /\* number is ok \*/

ungetch(c);

s[i] = '\0';

return (number);

} else { /\* it's too big; skip rest of line \*/

while (c != '\n' && c != EOF)

c = getchar();

s[lim-1] = '\0';

return (toobig);

}

}

Що ж являють собою функції 'getch' і 'ungetch'? Часто так буває, що програма, що зчитує вхідні дані, не може визначити, що вона прочитала вже досить, поки вона не прочитає занадто багато. Одним із прикладів є вибір символів, складових число: поки не з'явиться символ, відмінний від цифри, число не закінчено. Але при цьому програма зчитує один зайвий символ, символ, для якого вона ще не підготовлена.

Ця проблема була б вирішена, якби було б можливо "прочитати назад" небажаний символ. Тоді кожен раз, прочитавши зайвий символ, програма могла б помістити його назад в файл введення таким чином, що інша частина програми могла б вести себе так, немов цей символ ніколи не зчитувався. На щастя, таке неотримання символу легко імітувати, написавши пару діють спільно функцій. Функція getch доставляє наступний символ вводу, що підлягає розгляду; функція ungetch поміщає символ назад у введення, так що при наступному зверненні до getch він буде повернутий.

Те, як ці функції спільно працюють, досить просто. Функція ungetch поміщає повертаються назад символи в спільно використовуваний буфер, який є символьним масивом. Функція getch читає з цього буфера, якщо в ньому що-небудь є; якщо ж буфер порожній, вона звертається до getchar. При цьому також потрібна індексує змінна, яка буде фіксувати позицію поточного символу в буфері.

Так как буфер и его индекс совместно используются функциями getch и ungetch и должны сохранять свои значения в период между обращениями, они должны быть внешними для обеих функций. Таким образом, мы можем написать getch, ungetch и эти переменные как:

#define bufsize 100

char buf[bufsize]; /\* buffer for ungetch \*/

int bufp = 0; /\* next free position in buf \*/

getch() /\* get a (possibly pushed back) character \*/

{

return((bufp > 0) ? buf[--bufp] : getchar());

}

ungetch(c) /\* push character back on input \*/

int c;

{

if (bufp > bufsize)

printf("ungetch: too many characters\n");

else

buf [bufp++] = c;

}

Мы использовали для хранения возвращаемых символов массив, а не отдельный символ, потому что такая общность может пригодиться в дальнейшем.

**Упражнение 4-4**

Напишите функцию ungets(s), которая будет возвращать во ввод целую строку. Должна ли ungets иметь дело с buf и bufp или она может просто использовать ungetch?

**Упражнение 4-5**

Предположите, что может возвращаться только один символ. Измените getch и ungetch соответствующим образом.

**Упражнение 4-6**

Наши функции getch и ungetch не обеспечивают обработку возвращенного символа EOF переносимым образом. Решите, каким свойством должны обладать эти функции, если возвращается EOF, и реализуйте ваши выводы.

#### 4.6. Статические переменные

Статические *переменные* представляют собой третий *класс памяти*, в дополнении к *автоматическим переменным* и extern, с которыми мы уже встречались.

Статические *переменные* могут быть либо внутренними, либо внешними. Внутренние статические *переменные* точно так же, как и *автоматические*, являются локальными для некоторой *функции*, но, в отличие от *автоматических*, они остаются существовать, а не появляются и исчезают вместе с обращением к этой *функции*. это означает, что внутренние статические *переменные* обеспечивают постоянное, недоступное извне хранение внутри *функции*. *Символьные строки*, появляющиеся внутри *функции*, как, например, *аргументы* printf, являются внутренними статическими.

*Внешние статические переменные* определены в остальной части того *исходного файла*, в котором они описаны, но не в каком-либо другом файле. Таким образом, они дают способ скрывать имена, подобные buf и bufp в комбинации getch - ungetch, которые в силу их совместного использования должны быть внешними, но все же не доступными для пользователей getch и ungetch, чтобы исключалась возможность конфликта. Если эти две *функции* и две *переменные* объединить в одном файле следующим образом

static char buf[bufsize]; /\* buffer for ungetch \*/

static int bufp=0; /\*next free position in buf \*/

getch() {...}

ungetch() {...}

то никакая другая *функция* не будет в состоянии обратиться к buf и bufp ; фактически, они не будут вступать в конфликт с такими же именами из других файлов той же самой *программы*.

*Статическая память*, как внутренняя, так и внешняя, специфицируется словом static, стоящим перед обычным *описанием*. *Переменная является внешней*, если она описана вне какой бы то ни было *функции*, и внутренней, если она описана внутри некоторой *функции*.

Нормально *функции* являются внешними объектами; их имена известны глобально. возможно, однако, объявить *функцию* как static ; тогда ее имя становится неизвестным вне файла, в котором оно описано.

В языке "C" " static " отражает не только постоянство, но и степень того, что можно назвать "приватностью". Внутренние *статические объекты* определены только внутри одной *функции* ; внешние *статические объекты* / *переменные* или *функции* / определены только внутри того *исходного файла*, где они появляются, и их имена не вступают в конфликт с такими же именами *переменных* и *функций* из других файлов.

*Внешние статические переменные* и *функции* предоставляют способ организовывать *данные* и работающие с ними внутренние *процедуры* таким образом, что другие *процедуры* и *данные* не могут прийти с ними в конфликт даже по недоразумению. Например, *функции* getch и ungetch образуют "модуль" для *ввода* и возвращения символов; buf и bufp должны быть статическими, чтобы они не были доступны извне. Точно так же *функции* push, pop и clear формируют модуль обработки *стека* ; var и sp тоже должны быть внешними статическими.

#### 4.7. Регистровые переменные

Четвертый и последний *класс памяти* называется регистровым. *описание* register указывает *компилятору*, что данная *переменная* будет часто использоваться. Когда это возможно, *переменные*, *описанные* как register, располагаются в *машинных регистрах*, что может привести к меньшим по размеру и более быстрым *программам*. *описание* register выглядит как

register int x;

register char c;

и т.д.; часть int может быть опущена. *описание* register можно использовать только для *автоматических переменных* и формальных параметров *функций*. В этом последнем случае *описания* выглядят следующим образом:

f(c,n)

register int c,n;

{

register int i;

...

}

На практике возникают некоторые ограничения на регистровые *переменные*, отражающие реальные возможности имеющихся аппаратных средств. В регистры можно поместить только несколько *переменных* в каждой *функции*, причем только *определенных* *типов*. В случае превышения возможного числа или использования неразрешенных *типов* слово register игнорируется. Кроме того невозможно извлечь *адрес* регистровой *переменной* (этот вопрос обсуждается в "лекции №5" ). Эти специфические ограничения варьируются от машины к машине. Так, например, на *PDP-11* эффективными являются только первые три *описания* register в *функции*, а в качестве *типов* допускаются int, char или *указатель*.

#### 4.8. Блочная структура

Язык "C" не является языком с блочной *структурой* в смысле PL/1 или алгола; в нем нельзя описывать одни *функции* внутри других.

*переменные* же, с другой стороны, могут определяться по методу блочного *структурирования*. *описания* *переменных* (включая инициализацию) могут следовать за левой *фигурной скобкой*,открывающей любой *оператор*, а не только за той, с которой начинается тело *функции*. *переменные*, *описанные* таким образом, вытесняют любые *переменные* из внешних блоков, имеющие такие же имена, и остаются *определенными* до соответствующей правой *фигурной скобки*. Например в

if (n > 0) {

int i; /\* declare a new i \*/

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

...

}

Областью действия *переменной* i является "истинная" ветвь if ; это i никак не связано ни с какими другими i в *программе*.

Блочная *структура* влияет и на область действия *внешних переменных*. Если даны *описания*

int x;

f()

{

double x;

...

}

То появление x внутри *функции* f относится к внутренней *переменной* *типа* double, а вне f - к *внешней целой переменной*. Это же справедливо в отношении имен формальных параметров:

int x;

f(x)

double x;

{

...

}

Внутри *функции* f имя x относится к формальному параметру, а не к *внешней переменной*.

#### 4.9. Инициализация

Мы до сих пор уже много раз упоминали инициализацию, но всегда мимоходом , среди других вопросов. Теперь, после того как мы обсудили различные классы памяти, мы в этом разделе просуммируем некоторые правила, относящиеся к инициализации.

Если явная инициализация отсутствует, то *внешним и статическим переменным* присваивается значение нуль; *автоматические* и регистровые *переменные* имеют в этом случае неопределенные значения (мусор).

Простые *переменные* (не *массивы* или *структуры* ) можно инициализировать при их *описании*, добавляя вслед за именем знак равенства и *константное* выражение:

int x = 1;

char squote = '\'';

long day = 60 \* 24; /\* minutes in a day \*/

Для *внешних и статических переменных* инициализация выполняется только один раз, на этапе компиляции. *Автоматические* и регистровые *переменные* инициализируются каждый раз при входе в *функцию* или блок. В случае *автоматических* и регистровых *переменных* *инициализатор* не обязан быть *константой*: на самом деле он может быть любым значимым выражением, которое может включать *определеные* ранее величины и даже *обращения к функциям*. Например, инициализация в *программе* *бинарного поиска* из "лекции №3" могла бы быть записана в виде

binary(x, v, n)

int x, v[], n;

{

int low = 0;

int high = n - 1;

int mid;

...

}

вместо

binary(x, v, n)

int x, v[], n;

{

int low, high, mid;

low = 0;

high = n - 1;

...

}

По своему результату, инициализации *автоматических переменных* являются сокращенной записью *операторов присваивания*. Какую форму предпочесть - в основном дело вкуса. мы обычно используем явные присваивания, потому что инициализация в *описаниях* менее заметна. Автоматические *массивы* не могут быть инициализированы. Внешние и статические *массивы* можно инициализировать, помещая вслед за *описанием* заключенный в *фигурные скобки* список начальных значений, разделенных запятыми. Например *программа* подсчета символов из "лекции №1" , которая начиналась с

main() /\* count digits, white space, others \*/

(

int c, i, nwhite, nother;

int ndigit[10];

nwhite = nother = 0;

for (i = 0; i < 10; i++)

ndigit[i] = 0;

...

)

может быть переписана в виде

int nwhite = 0;

int nother = 0;

int ndigit[10] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

main() /\* count digits, white space, others \*/

(

int c, i;

...

)

Эти инициализации фактически не нужны, так как все присваиваемые значения равны нулю, но хороший стиль - сделать их явными. Если количество начальных значений меньше, чем указанный размер *массива*, то остальные элементы заполняются нулями. Перечисление слишком большого числа начальных значений является ошибкой. К сожалению, не предусмотрена возможность указания, что некоторое начальное значение повторяется, и нельзя инициализировать элемент в середине *массив* без перечисления всех предыдущих.

Для *символьных массивов* существует специальный способ инициализации; вместо *фигурных скобок* и запятых можно использовать строку:

char pattern[] = "the";

Это сокращение более длинной, но эквивалентной записи:

char pattern[] = { 't', 'h', 'e', '\0' };

Если размер *массив* любого *типа* опущен, то *компилятор* определяет его длину, подсчитывая число начальных значений. В этом конкретном случае размер равен четырем (три символа плюс конечное \0 ).

#### 4.10. Рекурсия

В языке "C" функции могут использоваться рекурсивно; это означает, что функция может прямо или косвенно обращаться к себе самой. Традиционным примером является печать числа в виде строки символов. как мы уже ранее отмечали, цифры генерируются не в том порядке: цифры младших разрядов появляются раньше цифр из старших разрядов, но печататься они должны в обратном порядке.

Эту проблему можно решить двумя способами. Первый способ, которым мы воспользовались в "лекции №3" в функции itoa, заключается в запоминании цифр в некотором массиве по мере их поступления и последующем их печатании в обратном порядке. Первый вариант функции printd следует этой схеме.

printd(n) /\* print n in decimal \*/

int n;

{

char s[10];

int i;

if (n < 0) {

putchar('-');

n = -n;

}

i = 0;

do {

s[i++] = n % 10 + '0'; /\* get next char \*/

} while ((n /= 10) > 0); /\* discard it \*/

while (--i >= 0)

putchar(s[i]);

}

Альтернативой этому способу является рекурсивное решение, когда при каждом вызове функция printd сначала снова обращается к себе, чтобы скопировать лидирующие цифры, а затем печатает последнюю цифру.

printd(n) /\* print n in decimal (recursive)\*/

int n;

(

int i;

if (n < 0) {

putchar('-');

n = -n;

}

if ((i = n/10) != 0)

printd(i);

putchar(n % 10 + '0');

)

Когда функция вызывает себя рекурсивно, при каждом обращении образуется новый набор всех автоматических переменных, совершенно не зависящий от предыдущего набора. Таким образом, в printd(123) первая функция printd имеет n = 123. Она передает 12 второй printd, а когда та возвращает управление ей, печатает 3. Точно так же вторая printd передает 1 третьей (которая эту единицу печатает), а затем печатает 2.

Рекурсия обычно не дает никакой экономии памяти, поскольку приходится где-то создавать стек для обрабатываемых значений. Не приводит она и к созданию более быстрых программ. Но рекурсивные программы более компактны, и они зачастую становятся более легкими для понимания и написания. Рекурсия особенно удобна при работе с рекурсивно определяемыми структурами данных, например, с деревьями; хороший пример будет приведен в "лекции №6" .

**Упражнение 4-7**

Приспособьте идеи, использованные в printd для рекурсивного написания itoa ; т.е. Преобразуйте целое в строку с помощью рекурсивной процедуры.

**Упражнение 4-8**

Напишите рекурсивный вариант функции reverse(s), которая располагает в обратном порядке строку s.

#### 4.11. Препроцессор языка "C"

В языке "с" предусмотрены определеные расширения языка с помощью простого макропредпроцессора. одним из самых распространенных таких расширений, которое мы уже использовали, является конструкция #define; другим расширением является возможность включать во время компиляции содержимое других файлов.

##### 4.11.1. Включение файлов

Для облегчения работы с наборами конструкций #define и описаний (среди прочих средств) в языке "с" предусмотрена возможность включения файлов. Любая строка вида

#include "filename"

заменяется содержимым файла с именем filename. (Кавычки обязательны). Часто одна или две строки такого вида появляются в начале каждого исходного файла, для того чтобы включить общие конструкции #define и описания extern для глобальных переменных. Допускается вложенность конструкций #include.

конструкция #include является предпочтительным способом связи описаний в больших программах. Этот способ гарантирует, что все исходные файлы будут снабжены одинаковыми определениями и описаниями переменных, и, следовательно, исключает особенно неприятный сорт ошибок. Естественно, когда какой-то включаемый файл изменяется, все зависящие от него файлы должны быть перекомпилированы.

##### 4.11.2. Макроподстановка

определение вида

#define tes 1

приводит к макроподстановке самого простого вида - замене имени на строку символов. Имена в #define имеют ту же самую форму, что и идентификаторы в "с"; заменяющий текст совершенно произволен. Нормально заменяющим текстом является остальная часть строки; длинное определение можно продолжить, поместив \ в конец продолжаемой строки. "Область действия" имени, определенного в #define, простирается от точки определения до конца исходного файла. Имена могут быть переопределены, и определения могут использовать определения, сделанные ранее. Внутри заключенных в кавычки строк подстановки не производятся, так что если, например, yes - определенное имя, то в printf("yes") не будет сделано никакой подстановки.

Так как реализация #define является частью работы макропредпроцессора, а не собственно компилятора, имеется очень мало грамматических ограничений на то, что может быть определено. Так, например, любители алгола могут объявить

#define then

#define begin {

#define end ;}

и затем написать

if (i > 0) then

begin

a = 1;

b = 2

end

Имеется также возможность определения макроса с аргументами, так что заменяющий текст будет зависеть от вида обращения к макросу. Определим, например, макрос с именем max следующим образом:

#define max(a, b) ((a) > (b) ? (a) : (b))

когда строка

x = max(p+q, r+s);

будет заменена строкой

x = ((p+q) > (r+s) ? (p+q) : (r+s));

Такая возможность обеспечивает " функцию максимума", которая расширяется в последовательный код, а не в обращение к функции. При правильном обращении с аргументами такой макрос будет работать с любыми типами данных ; здесь нет необходимости в различных видах max для данных разных типов, как это было бы с функциями.

Конечно, если вы тщательно рассмотрите приведенное выше расширение max, вы заметите определеные недостатки. Выражения вычисляются дважды; это плохо, если они влекут за собой побочные эффекты, вызванные, например, обращениями к функциям или использованием операций увеличения. Нужно позаботиться о правильном использовании круглых скобок, чтобы гарантировать сохранение требуемого порядка вычислений. (Рассмотрите макрос

#define square(x) x \* x

при обращении к ней, как square(z+1). Здесь возникают даже некоторые чисто лексические проблемы: между именем макро и левой круглой скобкой, открывающей список ее аргументов, не должно быть никаких пробелов.

Тем не менее аппарат макросов является весьма ценным. Один практический пример дает описываемая в "лекции №7" стандартная библиотека ввода-вывода, в которой getchar и putchar определены как макросы (очевидно putchar должна иметь аргумент ), что позволяет избежать затрат на обращение к функции при обработке каждого символа.

Другие возможности макропроцессора описаны в "приложении А" .

**Упражнение 4-9**

Определите макрос swap(x, y), который обменивает значениями два своих аргумента типа int. (В этом случае поможет блочная структура ).