**Типи, операції и вирази**

Змінні і константи є основними об'єктами, з якими оперує програма. Описи перераховують змінні, які будуть використовуватися, вказують їх тип і, можливо, їх початкові значення. Операції визначають, що з ними буде зроблено. Вирази об'єднують змінні і константи для отримання нових значень. Все це - теми цієї лекції.

**2.1. Імена змінних**

Хоча ми цього відразу прямо не сказали, існують деякі обмеження на імена змінних і символічних констант. Імена складаються з літер і цифр; перший символ повинен бути буквою. Підкреслення "\_" теж вважається буквою; це корисно для зручності читання довгих імен змінних. Великі та малі літери розрізняються; традиційна практика в "з" - використовувати малі літери для імен змінних, а прописні - для символічних констант.

Грають роль тільки перші вісім символів внутрішнього імені, хоча використовувати можна і більше. Для зовнішніх імен, таких як імена функцій і зовнішніх змінних, це число може виявитися менше восьми, так як зовнішні імена використовуються різними Асемблерами і завантажувачами. Деталі наводяться в "додатку А". Крім того, такі ключові слова як if, else, int, float і т.д., зарезервовані: ви не можете використовувати їх в якості імен змінних. (Вони пишуться малими літерами).

Звичайно, розумно вибирати імена змінних таким чином, щоб вони означали щось, що відноситься до призначення змінних, і щоб було менш імовірно сплутати їх при написанні.

**2.2. Типи і розміри даних**

У мові C є тільки кілька основних типів даних: char один байт, в якому може знаходитися один символ з внутрішнього набору символів. int ціле, зазвичай відповідне природному розміром цілих в використовуваної машині. float з плаваючою крапкою одинарної точності. double з плаваючою краркою подвійної точності.

Крім того є ряд кваліфікаторів, які можна використовувати з типом int: short (короткий), long (довге) і unsigned (без знака). Кваліфікатори short і long вказують на різні розміри цілих. Числа без знаку підпадають під дію законів арифметики по модулю 2 певною мірою n, де n - число бітів в int; числа без знаків завжди невід'ємні. Описи з кваліфікаторами мають вигляд:

short int x;

long int y;

unsigned int z;

Cлово int в таких ситуаціях може бути опущено, що зазвичай і робиться.  
Кількість бітів, що відводяться під ці об'єкти залежить від наявної машини; в таблиці нижче наведені деякі характерні значення.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Таблиця 2.1. | | | | |
|  | **DEC** **PDP-11** | **HONEYWELL 6000** | **IBM 370** | **INTERDATA 8/32** |
|  | ASCII | ASCII | EBCDIC | ASCII |
| char | 8-BITS | 9-BITS | 8-BITS | 8-BITS |
| int | 16 | 32 | 32 | 32 |
| short | 16 | 36 | 16 | 16 |
| long | 32 | 36 | 32 | 32 |
| float | 32 | 36 | 32 | 32 |
| double | 64 | 72 | 64 | 64 |

Мета полягає в тому, щоб short і long давали можливість в залежності від практичних потреб використовувати різні довжини цілих; тип int відображає найбільш "природний" розмір конкретної машини. Як ви бачите, кожен компілятор вільно інтерпретує short і long відповідно до своїх апаратних засобів. Все, на що ви можете твердо покладатися, це те, що short не довше, ніж long.

**2.3. Константи**

Константи типу int і float ми вже розглянули. Відзначимо ще тільки, що як звичайна  
123.456е-7,  
так і "науковий" запис  
0.12е3  
для float є законним.

Кожна константа з плаваючою крапкою вважається має тип double, так що позначення "e" служить як для float, так і для double.  
Довгі константи записуються у вигляді 123L. Звичайна ціла константа, яка дуже довга для типу int, розглядається як long.

Існує система позначень для вісімкових і шістнадцяткових констант: лідируючий 0 (нуль) в константі типу int вказує на восьмеричну константу, а стоячі попереду 0x відповідають шестнадцатеричной константі. Наприклад, десяткове число 31 можна записати як 037 в вісімковій формі і як 0x1f в шістнадцятковій. Шістнадцятиричні і восьмеричні константи можуть також закінчуватися буквою l, що робить їх відносними до типу long.

**2.3.1. Символьна константа**

**Символьна константа** - це один символ, укладений в одинарні лапки, як, наприклад, 'х'. Значенням символьної константи є чисельне значення цього символу у внутрішньому машинному наборі символів. Наприклад, в наборі символів ASCII символьний нуль, або "0", має значення 48, а в коді ebcdic - 240, і обидва ці значення зовсім відмінні від числа 0. Написання '0' замість чисельного значення, такого як 48 або 240, робить програму не залежить від конкретного чисельного уявлення цього символу в даній машині. Символьні константи точно так само беруть участь в численних операціях, як і будь-які інші числа, хоча найбільш часто вони використовуються в порівнянні з іншими символами. Правила перетворення будуть викладені пізніше.

Деякі неграфічні символи можуть бути представлені як символьні константи за допомогою умовних послідовностей, як, наприклад,

• \ n (новий рядок),  
• \ t (табуляція),  
• \ 0 (нульовий символ),  
• \\ (зворотна коса риска),  
• \ '(одинарна лапка) і т.д.

Хоча вони виглядають як два символи, насправді є одним. Крім того, можна згенерувати довільну послідовність двійкових знаків розміром в байт, якщо написати

'\ddd'

де ddd - від однієї до трьох вісімкових цифр, як в

#define formfeed '\014' /\* form feed \*/

Символьна константа '\ 0', що зображає символ із значенням 0, часто записується замість цілої константи 0, щоб підкреслити символьну природу деякого виразу.

##### 2.3.2. Константний вираз

*Константне вираз -* це вираз, що складається з одних констант. Такі вирази обробляються під час компіляції, а не при прогоні програми, і відповідно можуть бути використані в будь-якому місці, де можна використовувати константу, як, наприклад в

#define maxline 1000

char line[maxline+1];

або

seconds = 60 \* 60 \* hours;

##### 2.3.3. Рядкова константа

***Рядкова константа -*** це послідовність, що складається з нуля або більше символів, взятих в подвійні лапки, як, наприклад,

"i am a string" /\* я - рядок \*/ або

"" /\* null string \*/ /\* нуль-рядок \*/

Лапки не є частиною рядка, а служать тільки для її обмеження. Ті ж самі умовні послідовності, які використовувалися в символьних константах, застосовуються і в рядках; символ подвійної лапки зображується як \".

З технічної точки зору рядок являє собою масив, елементами якого є окремі символи. Щоб програмі було зручно визначати кінець рядка, компілятор автоматично поміщає в кінець кожного рядка нуль-символ \ 0. Таке уявлення означає, що ні накладається конкретного обмеження на те, яку довжину може мати рядок, і щоб визначити цю довжину, програми повинні переглядати рядок повністю. При цьому для фізичного зберігання рядка потрібно на одну комірку пам'яті більше, ніж число укладених в лапки символів. Наступна функція strlen (s) обчислює довжину символьного рядка s не рахуючи кінцевий символ \0.

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

char s[];

{

int i;

i = 0;

while (s[i] != '\0')

++i;

return(i);

}

Будьте уважні і не плутайте символьну константу з рядком, що містить один символ: 'x' - це не те ж саме, що "x". Перше - це окремий символ, використаний з метою отримання чисельного значення, відповідного букві х в машинному наборі символів. Друге - символьний рядок, що складається з одного символу (буква х) і \0.

#### 2.4. Опис

Всі змінні повинні бути описані до їх використання, хоча деякі описи робляться неявно, по контексту. Опис складається з специфікатору типу і наступного за ним списку змінних, що мають цей тип, як, наприклад,

int lower, upper, step;

char c, line[1000];

змінні можна розподіляти за описами будь-яким чином; наведені вище списки можна з тим же успіхом записати у вигляді

int lower;

int upper;

int step;

char c;

char line[1000];

Така форма займає більше місця, але вона зручна для додавання коментарів до кожного опису і для наступних модифікацій.

Змінним можуть бути присвоєні початкові значення всередині їх опису, хоча тут є деякі обмеження. Якщо за ім'ям змінної слідують знак рівності і константа, то ця константа служить в якості ініціалізатор, як, наприклад, в

char backslash = '\\';

int i = 0;

float eps = 1.0e-5;

Якщо розглянута змінна є зовнішньою або статичною, то ініціалізація проводиться тільки один раз, згідно з концепцією до початку виконання програми. Ініціалізуємих явно автоматичним змінним початкові значення присвоюються при кожному зверненні до функції, в якій вони описані. Автоматичні змінні, що не ініціалізуємі явно, мають невизначені значення, (тобто сміття). Зовнішні і статичні змінні за замовчуванням не започатковано нулем, але, тим не менше, їх явна ініціалізація є ознакою хорошого стилю.

Ми продовжимо обговорення питань ініціалізації, коли будемо описувати нові типи даних.

#### 2.5. Арифметичні операції

Бінарними арифметичними операціями є +, -, \*, / і операція ділення по модулю %. Є унарна операція -, але не існує унарної операції +.

При розподілі цілих дрібна частина відкидається. Вираз

x % y

дає залишок від ділення x на y і, отже, дорівнює нулю, коли x ділиться на y точно. Наприклад, рік є високосним, якщо він ділиться на 4, але не ділиться на 100, виключаючи те, що діляться на 400 роки теж є високосними. Тому

if(year % 4 == 0 && year % 100 != 0 || year % 400 == 0)

рік високосний

else

рік не високосний

Операцію % не можна використовувати з типами float або double.

Операції + і - мають однакове старшинство, яке молодше однакового рівня старшинства операцій \*, / і %, які в свою чергу молодше унарного мінуса. Арифметичні операції групуються зліва направо. (Відомості про старшинство і асоціативності всіх операцій зібрані в таблиці в кінці цієї лекції). Порядок виконання асоціативних і комутативний операцій типу + і - не фіксується; компілятор може перегруповувати навіть закриті в круглі дужки вираження, пов'язані такими операціями. Таким чином, а + (b + c) може бути обчислено як (a + b) + c. Це рідко призводить до будь-якого розбіжності, але якщо необхідно забезпечити строго певний порядок, то потрібно використати наявні проміжні змінні.

Дії, що вживаються при переповненні і антипереповненні (тобто при отриманні занадто маленького по абсолютній величині числа), залежать від використовуваної машини.

#### 2.6. Операції відношення і логічні операції

#### Операціями відношення являються

>= > =< <

всі вони мають однакове старшинство. Безпосередньо за ними за рівнем старшинства слідують операції рівності і нерівності:

== !=

які теж мають однакове старшинство. Операції відношення молодше арифметичних операцій, так що висловлювання на кшталт i <lim-1 розуміються як i <(lim-1), як і передбачається.

Логічні зв'язки && і || цікавіші. Вирази, пов'язані операціями && і ||, обчислюються зліва направо, причому їх розгляд припиняється відразу ж як тільки стає ясно, чи буде результат істиною або брехнею. Облік цих властивостей дуже істотний для написання правильно працюючих програм. Розглянемо, наприклад, оператор циклу в зчитує рядок функції getline, яку ми написали в "лекции №1" .

for(i=0;i<lim-1 && (c=getchar())

!= '\n' && c != EOF; ++i)

s[i]=c;

Ясно, що перед зчитуванням нового символу необхідно перевірити, чи є ще місце в масиві s, так що умова i <lim-1 має перевірятися першим. І якщо ця умова не виконується, ми не повинні зчитувати наступний символ.

Так само невдалим було б порівняння 'c' з EOF до звернення до функції getchar: перш ніж перевіряти символ, його потрібно вважати.

Старшинство операції && вище, ніж у ||, і обидві вони молодші операцій відносини і рівності. Тому такі вирази, як

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

не потребують додаткових круглих дужках. Але так як операція! = Старше операції привласнення, то для досягнення правильного результату в вираженні

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

Дужки необхідні.

Унарна операція заперечення ! Перетворює ненульовий або істинний операнд в 0, а нульовий або помилковий операнд в 1. Звичайне використання операції ! Полягає в запису

if( ! inword )

Замість

if( inword == 0 )

Важко сказати, яка форма краще. конструкції типу! inword Читаються досить зручно ("якщо не в слові"). Але в більш складних випадках вони можуть виявитися важкими для розуміння.

**Вправа 2-1**

Напишіть оператор циклу, еквівалентний наведеним вище оператору for, не використовуючи операції &&.

#### 2.7. Перетворення типів

Якщо у виразах зустрічаються операнди різних типів, то вони перетворюються до загального типу відповідно до невеликим набором правил. Загалом, автоматично виробляються тільки перетворення, які мають сенс, такі як, наприклад, перетворення цілого в плаваюче в виразах типу f + i. Вирази ж, позбавлені сенсу, такі як використання змінної типу float в якості індексу, заборонені.

По-перше, типи char і int можуть вільно змішуватися в арифметичних виразах: кожна змінна типу char автоматично перетвориться в int. Це забезпечує значну гнучкість при проведенні певних перетворень символів. Прикладом може служити функція atoi, яка ставить у відповідність рядку цифр її чисельний еквівалент.

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

char s[];

{

int i, n;

n = 0;

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

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

return(n);

}

Kак вже обговорювалося в "лекції №1", вираз

s[i] - '0'

має чисельне значення знаходиться в s [i] символу, тому що значення символів '0', '1' і т.д. утворюють зростаючу послідовність розташованих підряд цілих позитивних чисел.  
Інший приклад перетворення char в int дає функція lower, яка перетворює цю прописну букву в малу. Якщо виступаючий в якості аргументу символ не є великою літерою, то lower повертає його незмінним. Наведена нижче програма справедлива тільки для набору символів ASCII.

lower(c) /\* convert c to lower case; ascii only \*/

int c;

{

if ( c >= 'a' && c <= 'z' )

return( c + '@' - 'a');

else /\*@ Записано замість 'a' строчного\*/

return(c);

}

Ця функція правильно працює при коді ASCII, тому що чисельні значення, відповідні в цьому коді прописних і рядкових буквах, відрізняються на постійну величину, а кожен алфавіт є суцільним - між а і z немає нічого, крім букв. Це останнє зауваження для набору символів ebcdic систем IBM 360/370 виявляється несправедливим, в силу чого ця програма на таких системах не працює належним чином - вона перетворює не тільки букви.

При перетворенні символьних змінних в цілі виникає один тонкий момент. Справа в тому, що сама мова не вказує, чи повинні змінним типу char відповідати чисельні значення зі знаком або без знаку. Чи може при перетворенні char в int вийти негативне ціле? На жаль, відповідь на це питання змінюється від машини до машини, відображаючи розбіжності в їх архітектурі. На деяких машинах (PDP-11, наприклад) змінна типу char, крайній лівий біт якої містить 1, перетворюється в негативне ціле ( "знакове розширення"). На інших машинах таке перетворення супроводжується додаванням нулів з лівого краю, в результаті чого завжди виходить позитивне число.

визначення мови "C" гарантує, що будь-який символ з стандартного набору символів машини ніколи не дасть негативного числа, так що ці символи можна вільно використовувати у виразах як позитивні величини. Але довільні комбінації двійкових знаків, що зберігаються як символьні змінні на деяких машинах, можуть дати негативні значення, а на інших позитивні.

Найбільш типовим прикладом виникнення такої ситуації є випадок, коли значення -1 використовується в якості EOF. Розглянемо програму

char c;

c = getchar();

if ( c == EOF)

...

На машині, яка здійснює знакового розширення, змінна 'с' завжди позитивна, оскільки вона описана як char, а так як EOF негативно, то умова ніколи не виконується. Щоб уникнути такої ситуації, ми завжди завбачливо використали int замість char для будь-якої змінної, яка отримує значення від getchar.

Основна ж причина використання int замість char не пов'язана з будь-яким питанням про можливе знакове розширення. просто функція getchar повинна передавати всі можливі символи (щоб її можна було використовувати для довільного введення) і, крім того, що відрізняється значення EOF. Отже значення EOF не може бути представлено як char, а повинно зберігатися як int.

Іншою корисною формою автоматичного перетворення типів є те, що вираження ставлення, подібні i> j, і логічні вираження, пов'язані операціями && і ||, за визначенням мають значення 1, якщо вони істинні, і 0, якщо вони помилкові. Таким чином, присвоєння

isdigit = c >= '0' && c <= '9';

вважає isdigit рівним 1, якщо с - цифра, і рівним 0 в іншому випадку. (В перевірочній частини операторів if, while, for і т.д. "Істинно" просто означає «не нуль").

Неявні арифметичні перетворення працюють в основному, як і очікується. У загальних рисах, якщо операція типу + або \*, яка пов'язує два операнда (бінарна операція), має операнди різних типів, то перед виконанням операції "нижчий" тип перетворюється до "високого" і виходить результат "вищого" типу. Більш точно, до кожної арифметичної операції застосовується наступна послідовність правил перетворення.

* типи char і short перетворюються в int, а float в double.
* Потім, якщо один з операндів має тип double, то інший перетвориться в double, і результат має тип double.
* В іншому випадку, якщо один з операндів має тип long, то інший перетвориться в long, і результат має тип long.
* В іншому випадку, якщо один з операндів має тип unsigned, то інший перетвориться в unsigned і результат має тип unsigned.
* В іншому випадку операнди повинні бути типу int, і результат має тип int. Підкреслимо, що всі змінні типу float у виразах перетворюються в double; в "C" вся плаваюча арифметика виконується з подвійною точністю.

Перетворення виникають і при привласнення; значення правої частини перетвориться до типу лівої, який і є типом результату. Символьні змінні перетворюються в цілі або зі знаковим розширенням, або без нього, як описано вище. Зворотне перетворення int в char поводиться добре - зайві біти високого порядку просто відкидаються. Таким чином

int i; char c;

i = c; c = i;

значення 'с' не змінюється. Це вірно незалежно від того, залучається чи знакове розширення чи ні.  
Якщо х типу float, а i типу int, то як

х = i; так як

i = х;

призводять до перетворень; при цьому float перетворюється в int відкиданням дробової частини. тип double перетворюється на float округленням. Довгі цілі перетворюються в більш короткі цілі і в змінні типу char за допомогою відкидання зайвих бітів високого порядку.

Так як аргумент функції є виразом, то при передачі функцій аргументів також відбувається перетворення типів: зокрема, char і short стають int, а float стає double. Саме тому ми описували аргументи функцій як int і double навіть тоді, коли зверталися до них зі змінними типу CHAR і float.

Нарешті, в будь-якому вираженні може бути здійснено ("змушений") явне перетворення типу за допомогою конструкції, іменованої переклад (cast). У цій конструкції, що має вигляд  
(Ім'я типу) вираз

Вираз перетвориться до зазначеного типу за правилами перетворення, викладеними вище. Фактично точний зміст операції переказу можна описати таким чином: вираз як би присвоюється деякій змінній зазначеного типу, яка потім використовується замість всієї конструкції. Наприклад, бібліотечна процедура sqrt очікує аргументу типу double і видасть безглуздий відповідь, якщо до неї через недбалість звернуться з чим-небудь іншим. таким чином, якщо n - ціле, то вираз

sqrt((double) n)

до передачі аргументу функції sqrt перетворює n до типу double. (Відзначимо, що операція переклад перетворює значення n в належний тип; фактичний зміст змінної n при цьому не змінюється). Операція переказу має той же рівень старшинства, що і інші унарні операції, як вказується в таблиці в кінці цієї лекції.

**Вправа 2-2**Складіть програму для функції htoi (s), яка перетворює рядок шістнадцятирічних цифр в еквівалентну їй ціле значення. При цьому допустимими цифрами є цифри від 1 до 9 і букви від а доF.

#### 2.8. Операції збільшення і зменшення

У мові "C" передбачені дві незвичайні операції для збільшення і зменшення значень змінних. Операція збільшення ++ додає 1 до свого операнду, а операція зменшення - віднімає 1. Ми часто використовували операцію ++ для збільшення змінних, як, наприклад, в

if(c == '\n')

++i;

Незвичайний аспект полягає в тому, що ++ і - можна використовувати або як префіксні операції (перед змінної, як в ++ n), або як постфіксні (після змінної: n ++). Ефект в обох випадках полягає в збільшенні n. Але вираз ++ n збільшує змінну n до використання її значення, в той час як n ++ збільшує змінну n після того, як її значення було використано. Це означає, що в контексті, де використовується значення змінної, а не тільки ефект збільшення, використання ++ n і n ++ призводить до різних результатів. якщо n = 5, то

х = n++;

встановлює х рівним 5, а

х = ++n;

вважає х рівним 6. В обох випадках n стає рівним 6. Операції збільшення і зменшення можна застосовувати тільки до змінних; вирази типу х = (i + j) ++ є незаконними.

У випадках, де потрібен тільки ефект збільшення, а саме значення не використовується, як, наприклад, в

if ( c == '\n' )

nl++;

вибір префіксної або постфіксной операції є справою смаку. Але зустрічаються ситуації, де потрібно використовувати саме ту або іншу операцію. Розглянемо, наприклад, функцію squeeze (s, c), яка видаляє символ 'с' з рядка s, кожен раз, як він зустрічається.

squeeze(s,c) /\* delete all c from s \*/

char s[];

int c;

{

int i, j;

for ( i = j = 0; s[i] != '\0'; i++)

if ( s[i] != c )

s[j++] = s[i];

s[j] = '\0';

}

Кожен раз, як зустрічається символ, відмінний від 'с', він копіюється в поточну позицію j, і тільки після цього j збільшується на 1, щоб бути готовим до вступу наступного символу. Це в точності еквівалентно запису

if ( s[i] != c ) {

s[j] = s[i];

j++;

}

Інший приклад подібної конструкції дає функція getline, яку ми запрограмували в "лекції №1", де можна замінити

if ( c == '\n' ) {

s[i] = c;

++i;

}

Більш компактний запис

if ( c == '\n' )

s[i++] = c;

В якості третьої прикладу розглянемо функцію strcat (s, t), яка приписує рядок t в кінець рядка s, утворюючи конкатенацію рядків s і t. При цьому передбачається, що в s досить місця для зберігання отриманої комбінації.

strcat(s,t) /\* concatenate t to end of s \*/

char s[], t[]; /\* s must be big enough \*/

{

int i, j;

i = j = 0;

while (s[i] != '\0') / \*find end of s \*/

i++;

while((s[i++] = t[j++]) != '\0') /\*copy t\*/

;

}

Tак як з t в s копіюється кожен символ, то для підготовки до наступного проходженню циклу Постфіксний операція ++ застосовується до обох змінним i і j.

**Вправа 2-3**

Напишіть інший варіант функції squeeze (s1, s2), який видаляє з рядка s1 кожен символ, що співпадає з будь-яким символом рядка s2.

**Вправа 2-4**

Напишіть програму для функції any (s1, s2), яка знаходить місце першої появи в рядку s1 будь-якого символу з рядка s2 і, якщо рядок s1 не містить символів рядка s2, повертає значення -1.

#### 2.9.Побітові логічні операції

У мові передбачений ряд операцій для роботи з бітами; ці операції не можна застосовувати до змінних типу float або double.

|  |  |
| --- | --- |
| Таблиця 2.2. | |
| & | побітове AND |
| | | Побітове включаюче OR |
| ^ | Побітове виключаюче OR |
| << | Зсув вліво |
| >> | Зсув вправо |
| ~ | доповнення (унарна операція) |

" | " імітує вертикальну риску.

Побітова операція and часто використовується для маскування деякої множини бітів; наприклад, *оператор*

c = n & 007f

передає в 'с' сім молодших бітів n, вважаючи інші рівними нулю. Операція '|' побітового or використовується для включення бітів:

c = x | mask

встановлює на одиницю ті біти в х, які дорівнюють одиниці в mask.

Слід бути уважним і відрізняти побітові операції & і | від логічних зв'язок && і ||, які мають на увазі обчислення значення істинності зліва направо. Наприклад, якщо х = 1, а y = 2, то значення х & y дорівнює нулю, в той час як значення x && y дорівнює одиниці.

Операції зсуву << і >> здійснюють відповідно зсув вліво і вправо свого лівого операнда на число бітових позицій, що задаються правим операндом. Таким чином, х << 2 зсуває х вліво на дві позиції, заповнюючи звільняються біти нулями, що еквівалентно множенню на 4. Зсув вправо величини без знака заповнює звільняються біти на деяких машинах, таких як PDP-11, заповнюються вмістом знакового біта "арифметичне зрушення ", а на інших - нулем" логічний зрушення ".

Унарна операція ~ дає доповнення до цілого; це означає, що кожен біт зі значенням 1 отримує значення 0 і навпаки. Ця операція зазвичай виявляється корисною в виразах типу

x & ~077

де останні шість бітів х маскуються нулем. Підкреслимо, що вираз x &! 077 не залежить від довжини слова і тому краще, ніж, наприклад, x & 0177700, де передбачається, що х займає 16 бітів. Така форма що переноситься не вимагає ніяких додаткових витрат, оскільки ~ 077 є сталою виразом і, отже, обробляється під час компіляції.

Щоб проілюструвати використання деяких операцій з бітами, розглянемо функцію getbits (x, p, n), яка повертає / зсунутими до правого краю / починаються з позиції р поле змінної х довжиною n бітів. Ми припускаємо, що крайній правий біт має номер 0, і що n і р - розумно задані позитивні числа. Наприклад, getbits (х, 4,3) повертає зсунутими до правого краю біти, що займають позиції 4, 3 і 2.

getbits(x,p,n) /\* get n bits from position p \*/

unsigned x, p, n;

{

return((x >> (p+1-n)) & ~(~0 << n));

}

Операція x >> (p + 1-n) зсовує бажане поле в правий кінець слова. Опис аргументу x як unsigned гарантує, що при зсуві вправо звільняються біти будуть заповнюватися нулями, а не вмістом знакового біта, незалежно від того, на якій машині пропускається програма. Всі біти константного виразу ~ 0 рівні 1; зсув його на n позицій вліво з допомогою операції ~ 0 << n створює маску з нулями в n крайніх правих бітах і одиницями в інших; доповнення ~ створює маску з одиницями в n крайніх правих бітах.

**Вправа 2-5**

Переробіть getbits таким чином, щоб біти відлічувалися зліва направо.

**Вправа 2-6**

Напишіть програму для функції wordlength (), що обчислює довжину слова використовуваної машини, тобто число бітів у змінній типу int. функція повинна мати змогу переноситись ,тобто одна і та ж вихідна програма повинна правильно працювати на будь-якій машині.

**Вправа 2-7**

Ннапишіть програму для функції rightrot (n, b), зрушує циклічно ціле n вправо на b бітових позицій.

**Вправа 2-8**

Напишіть програму для функції invert (x, p, n), яка інвертує (тобто замінює 1 на 0 і навпаки) n бітів x, що починаються з позиції p, залишаючи інші біти незміненими.

#### 2.10. Операції і вирази присвоєння

Такі вирази як

i = i + 2

в яких ліва частина повторюється в правій частині можуть бути записані в стислій формі

i += 2

використовуючи операцію присвоювання виду +=.

Більшості бінарних операцій (операцій подібних +, які мають лівий і правий операнд) відповідає операція присвоювання виду оп =, де оп - одна з операцій + - \* / % << >> & ^ |!

Якщо е1 і е2 - вирази, то е1 оп= е2 еквівалентно

е1 = (е1) оп (е2)

за винятком того, що вираз е1 обчислюється тільки один раз. Зверніть увагу на круглі дужки навколо е2:

x \*= y + 1

то

x = x \* (y + 1)

не

x = x \* y + 1

Як приклад наведемо функцію bitcount, яка підраховує число рівних 1 бітів у цілого аргументу.

bitcount(n) /\* count 1 bits in n \*/

unsigned n;

(

int b;

for (b = 0; n != 0; n >>= 1)

if (n & 01)

b++;

return(b);

)

Не кажучи вже про стислості, такі оператори присвоювання мають ту перевагу, що вони краще відповідають образу людського мислення. Ми говоримо: "додати 2 до i" або "збільшити i на 2", але не "взяти i, додати 2 і помістити результат знову в i". Отже, i + = 2. Крім того, в громіздких виразах, подібних

yyval[yypv[p3+p4] + yypv[p1+p2]] += 2

Tакая операція присвоювання полегшує розуміння програми, так як читач не повинен скрупульозно перевіряти, чи є два довгих вирази дійсно однаковими, або замислюватися, чому вони не збігаються. Така операція присвоювання може навіть допомогти компілятору отримати більш ефективну програму.

Ми вже використовували той факт, що операція присвоєння має деяке значення і може входити у вираз; найтиповіший приклад

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

присвоювання, що використовують інші операції присвоєння (+ =, - = і т.д.) також можуть входити у вираз, хоча це трапляється рідше. Типом виразу присвоювання є тип його лівого операнда.

**Вправа 2-9**

У двійковій системі числення операція x & (x-1) обнуляє найправіший рівний 1 біт змінної x. (Чому?) Використовуйте це зауваження для написання більш швидкої версії функції bitcount.

#### 2.11. Умовні вирази

*оператори*

if (a > b)

z = a;

else

z = b;

звичайно обчислюють в z максимум з а й в. Умовний вираз, записаний за допомогою тернарной операції "?:", надає іншу можливість для запису цієї і аналогічних конструкцій. У вираженні

е1 ? Е2 : е3

спочатку обчислюється вираз е1. Якщо воно відмінно від нуля (істинно), то обчислюється вираз е2, яке і стає значенням умовного виразу. В іншому випадку обчислюється е3, і воно стає значенням умовного виразу. Щоразу обчислюється тільки одне з виразу е2 і е3. Таким чином, щоб покласти z рівним максимуму з а й в, можна написати

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

Слід підкреслити, що умовний вираз дійсно є виразом і може використовуватися точно так же, як будь-яке інше вираження. Якщо е2 і е3 мають різні типи, то тип результату визначається за правилами перетворення, розглянутим раніше в цій лекції. Наприклад, якщо f має тип float, а n - тип int, то вираз

(n > 0) ? f : n

Має тип double незалежно від того, позитивне n чи ні.

Так як рівень старшинства операції ?: дуже низький, прямо над присвоєнням, то перший вираз в умовному виразі можна не закривати в круглі дужки. Однак, ми все ж рекомендуємо це робити, так як дужки роблять умовну частину виразу більш помітною.

Використання умовних виразів часто призводить до коротких програмам. Наприклад, наступний нижче оператор циклу друкує n елементів масиву, по 10 в рядку, розділяючи кожен стовпчик одним пропуском і закінчуючи кожен рядок (включаючи останній) одним символом переведення рядка.

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

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

Символ перекладу рядка записується після кожного десятого елемента і після n -го елемента. За всіма іншими елементами слідує один пробіл. Хоча, можливо, це виглядає хитромудрою, було б повчальним спробувати записати це, не використовуючи умовного виразу.

**Вправа 2-10**

Перепишіть програму для функції lower, яка переводить великі літери на малі, використовуючи замість конструкції if - else умовний вираз.

#### 2.12. Старшинство і порядок обчислення

У наведеній нижче таблиці наведені правила старшинства і асоціативності всіх операцій, включаючи і ті, які ми ще не обговорювали. Операції, розташовані в одному рядку, мають один і той же рівень старшинства; рядки розташовані в порядку убування старшинства. Так, наприклад, операції \*, / і% мають однаковий рівень старшинства, який вище, ніж рівень операцій + і -.

|  |  |
| --- | --- |
| Таблиця 2.3. | |
| **OPERATOR** | **ASSOCIATIVITY** |
| () [] -> . | LEFT TO RIGHT |
| ! ^ ++ -- - (TYPE) \* & SIZEOF | RIGHT TO LEFT |
| \* / % | LEFT TO RIGHT |
| + - | LEFT TO RIGHT |
| << >> | LEFT TO RIGHT |
| < <= > >= | LEFT TO RIGHT |
| == != | LEFT TO RIGHT |
| & | LEFT TO RIGHT |
| ^ | LEFT TO RIGHT |
| | | LEFT TO RIGHT |
| && | LEFT TO RIGHT |
| || | LEFT TO RIGHT |
| ?: | RIGHT TO LEFT |
| = += -= ETC. | RIGHT TO LEFT |
| , (CHAPTER 3) | LEFT TO RIGHT |

Операції -> і. Використовуються для доступу до елементів структур; вони будуть описані в "лекції №6" разом з sizeof (розмір об'єкта). В "лекції №5" обговорюються операції \* (непряма адресація) і & (адреса). Відзначимо, що рівень старшинства побітових логічних операцій &, ^ і 'нижче рівня операцій == і! =. Це призводить до того, що здійснюють побітову перевірку вираження, подібні

if ((x & mask) == 0) ...

Для отримання правильних результатів повинні закривати в круглі дужки.

Як вже зазначалося раніше, вирази, в які входить одна з асоціативних і комутативний операцій (\*, +, &, ^, '), можуть перегруповуватися, навіть якщо вони укладені в круглі дужки. У більшості випадків це не призводить до яких би то не було розбіжностей; в ситуаціях, де такі розбіжності все ж можливі, для забезпечення потрібного порядку обчислень можна використовувати явні проміжні змінні.

У мові "C", як і в більшості мов, не фіксується порядок обчислення операндів в операторі. Наприклад в операторі вигляду

x = f() + g();

спочатку може бути обчислено f, а потім g, і навпаки; тому, якщо або f, або g змінюють зовнішню змінну, від якої залежить інший операнд, то значення x може залежати від порядку обчислень. Для забезпечення потрібної послідовності проміжні результати можна знову запам'ятовувати в тимчасових змінних.

Подібним же чином не фіксується порядок обчислення аргументів функції, так що оператор

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

може давати (і дійсно дає) на різних машинах різні результати в залежності від того, збільшується чи n до або після звернення до функції power. Правильним рішенням, звичайно, є запис

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

Звернення до функцій, вкладені операції привласнення, операції збільшення та зменшення призводять до так званих "побічних ефектів" - деякі змінні змінюються як побічний результат обчислення виразів. У будь-якому вираженні, в якому виникають побічні ефекти, можуть існувати дуже тонкі залежності від порядку, в якому визначаються входячі до нього змінні. Прикладом типової невдалої ситуації є оператор

a[i] = i++;

Виникає питання, старе або нове значення i служить в якості індексу. Компілятор може надходити різними способами і в залежності від своєї інтерпретації видавати різні результати. Той випадок, коли відбуваються побічні ефекти (привласнення фактичним змінним), - залишається на розсуд компілятора, так як найкращий порядок сильно залежить від архітектури машини.

З цих міркувань випливає така мораль: написання програм, що залежать від порядку обчислень, є поганим методом програмування на будь-якій мові. Звичайно, необхідно знати, чого слід уникати, але якщо ви не в курсі, як деякі речі реалізовані на різних машинах, це незнання може вберегти вас від неприємностей. (налагоджувальна програма lint вкаже більшість місць, що залежать від порядку обчислень).