Глава 2. Типы, операторы и выражения

Переменные и константы являются основными объектами данных, с которыми имеет дело программа. Переменные перечисляются в объявлениях, где устанавливаются их типы и, возможно, начальные значения. Операции определяют действия, которые совершаются с этими переменными. Выражения комбинируют переменные и константы для получения новых значений. Тип объекта определяет множество значений, которые этот объект может принимать, и операций, которые над ними, могут выполняться. Названные "кирпичики” и будут предметом обсуждения в этой главе.

Стандартом ANSI было утверждено значительное число небольших изменений и добавлений к основным типам и выражениям. Любой целочисленный тип теперь может быть со знаком, **signed**, и без знака, **unsigned**. Предусмотрен способ записи беззнаковых констант и шестнадцатеричных символьных констант. Операции с плавающей точкой допускаются теперь и с одинарной точностью. Введен тип **long double**, обеспечивающий повышенную точность. Строковые константы конкатенируются ("склеиваются”) теперь во время компиляции. Частью языка стали перечисления (**enum**), формализующие для типа установку диапазона значений. Объекты для защиты их от каких-либо изменений разрешено помечать как **const**. В связи с введением новых типов расширены правила автоматического преобразования из одного арифметического типа в другой.

2.1 Имена переменных

Хотя мы ничего не говорили об этом в главе 1, но существуют некоторые ограничения на задание имен переменных и именованных констант.

Имена составляются из букв и цифр; первым символом должна быть буква. Символ подчеркивания "_" считается буквой; его иногда удобно использовать, чтобы улучшить восприятие длинных имен переменных. Не начинайте имена переменных с подчеркивания, так как многие переменные библиотечных программ начинаются именно с этого знака. Большие (прописные) и малые (строчные) буквы различаются, так что x и X - это два разных имени. Обычно в программах на Си малыми буквами набирают переменные, а большими - именованные константы.

Для внутренних имен значимыми являются первые 31 символ. Для имен функций и внешних переменных число значимых символов может быть меньше 31, так как эти имена обрабатываются ассемблерами и загрузчиками и языком не контролируются. Уникальность внешних имен

гарантируется только в пределах 6 символов, набранных безразлично в каком регистре. Ключевые слова **if**, **else**, **int**, **float** и т. д. зарезервированы, и их нельзя использовать в качестве имен переменных. Все они набираются на нижнем регистре (т. е. малыми буквами).

Разумно давать переменным осмысленные имена в соответствии с их назначением, причем такие, чтобы их было трудно спутать друг с другом. Мы предпочитаем короткие имена для локальных переменных, особенно для счетчиков циклов, и более длинные для внешних переменных.

2.2 Типы и размеры данных

В Си существует всего лишь несколько базовых типов:

char - единичный байт, который может содержать один символ из допустимого символьного набора;
int - целое, обычно отображающее естественное представление целых в машине;
float - число с плавающей точкой одинарной точности;
double - число с плавающей точкой двойной точности.

Имеется также несколько квалификаторов, которые можно использовать вместе с указанными базовыми типами. Например, квалификаторы **short** (короткий) и **long** (длинный) применяются к целым:

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

В таких объявлениях слово **int** можно опускать, что обычно и делается. Если только не возникает противоречий со здравым смыслом, **short int** и **long int** должны быть разной длины, а **int** соответствовать естественному размеру целых на данной машине. Чаще всего для представления целого, описанного с квалификатором **short**, отводится 16 бит, с квалификатором **long** - 32 бита, а значению типа **int** - или 16, или 32 бита. Разработчики компилятора вправе сами выбирать подходящие размеры, сообразуясь с характеристиками своего компьютера и соблюдая следующие ограничения: значения типов **short** и **int** представляются по крайней мере 16 битами; типа **long** - по крайней мере 32 битами; размер **short** не больше размера **int**, который в свою очередь не больше размера **long**.

Квалификаторы **signed** (со знаком) или **unsigned** (без знака) можно применять к типу **char** и любому целочисленному типу. Значения **unsigned** всегда положительны или равны нулю и подчиняются законам арифметики по модулю 2^n, где n - количество бит в представлении типа. Так, если значению **char** отводится 8 битов, то **unsigned char** имеет значения в диапазоне от 0 до 255, а **signed char** – от -128 до 127 (в машине с двоичным дополнительным кодом). Являются ли значения типа просто **char** знаковыми или беззнаковыми, зависит от реализации, но в любом случае коды печатаемых символов положительны.

Тип **long double** предназначен для арифметики с плавающей точкой повышенной точности. Как и в случае целых, размеры объектов с плавающей точкой зависят от реализации; **float**, **double** и **long double** могут представляться одним размером, а могут - двумя или тремя разными размерами.

Именованные константы для всех размеров вместе с другими характеристиками машины и компилятора содержатся в стандартных заголовочных файлах <**limits.h**> и <**float.h**> (см. приложение B).

Упражнение 2.1. Напишите программу, которая будет выдавать диапазоны значений типов **char**, **short**, **int** и **long**, описанных как **signed** и как **unsigned**, с помощью печати соответствующих значений из стандартных заголовочных файлов и путем прямого вычисления. Определите диапазоны чисел с плавающей точкой различных типов. Вычислить эти диапазоны сложнее.

2.3 Константы

Целая константа, например 1234, имеет тип **int**. Константа типа **long** завершается буквой **l** или **L**, например 123456789L: слишком большое целое, которое невозможно представить как **int**, будет представлено как **long**. Беззнаковые константы заканчиваются буквой **u** или **U**, а окончание **ul** или **UL**

говорит о том, что тип константы - **unsigned long**.

Константы с плавающей точкой имеют десятичную точку (123.4), или экспоненциальную часть (1e-2), или же и то и другое. Если у них нет окончания, считается, что они принадлежат к типу **double**. Окончание **f** или **F** указывает на тип **float**, а **l** или **L** - на тип **long double**.

Целое значение помимо десятичного может иметь восьмеричное или шестнадцатеричное представление. Если константа начинается с нуля, то она представлена в восьмеричном виде, если с 0x или с 0X, то - в шестнадцатеричном. Например, десятичное целое 31 можно записать как 037 или как 0X1F. Записи восьмеричной и шестнадцатеричной констант могут завершаться буквой **L** (для указания на тип **long**) и **U** (если нужно показать, что константа беззнаковая). Например, константа 0XFUL имеет значение 15 и тип **unsigned long**.

Символьная константа есть целое, записанное в виде символа, обрамленного одиночными кавычками, например 'x'. Значением символьной константы является числовой код символа из набора символов на данной машине. Например, символьная константа '0' в кодировке ASCII имеет значение 48, которое никакого отношения к числовому значению 0 не имеет. Когда мы пишем '0' , а не какое-то значение (например 46), зависящее от способа кодировки, мы делаем программу независимой от частного значения кода, к тому же она и легче читается. Символьные константы могут участвовать в операциях над числами точно так же, как и любые другие целые, хотя чаще они используются для сравнения с другими символами.

Некоторые символы в символьных и строковых константах записываются с помощью эскейп-последовательностей, например \n (символ новой строки); такие последовательности изображаются двумя символами, но обозначают один. Кроме того, произвольный восьмеричный код можно задать в виде

```
'\ooo'
```

где ooo - одна, две или три восьмеричные цифры (0 ... 7) или

```
'\xhh'
```

где hh - одна, две или более шестнадцатеричные цифры (0...9, a...f, A...F). Таким образом, мы могли бы написать

```
#define VTAB '013' /* вертикальная табуляция в ASCII */
#define BELL '\007' /* звонок в ASCII */
```

или в шестнадцатеричном виде:

```
#define VTAB '\xb' /* вертикальная табуляций в ASCII */
#define BELL '\x7' /* звонок в ASCII */
```

Полный набор эскейп-последовательностей таков:

```
\a сигнал-звонок
\b возврат-на-шаг (забой)
\f перевод-страницы
\n новая-строка
\r возврат-каретки
\t горизонтальная-табуляция
\v вертикальная-табуляция
\\ обратная наклонная черта
\? знак вопроса
\' одиночная кавычка
\" двойная кавычка
\ooo восьмеричный код
\xhh шестнадцатеричный код
```

Символьная константа '\0' - это символ с нулевым значением, так называемый символ **null**. Вместо просто 0 часто используют запись '\0', чтобы подчеркнуть символьную природу выражения, хотя и в том и другом случае запись обозначает нуль.

Константные выражения - это выражения, оперирующие только с константами. Такие выражения вычисляются во время компиляции, а не во время выполнения, и поэтому их можно использовать в любом месте, где допустимы константы, как, например, в

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

или в

```
#define LEAP 1 /* in leap years - в високосные годы */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Строковая константа, или *строковый литерал*, - это нуль или более символов, заключенных в двойные кавычки, как, например,

```
"Я строковая константа”
```

или

```
"" /* пустая строка */
```

Кавычки не входят в строку, а служат только ее ограничителями. Так же, как и в символьные константы, в строки можно включать эскейп-последовательности; \", например, представляет собой двойную кавычку. Строковые константы можно конкатенировать ("склеивать”) во время компиляции; например, запись двух строк

```
"Здравствуй," " мир!"
```

эквивалентна записи одной следующей строки:

```
"Здравствуй, мир!"
```

Указанное свойство позволяет разбивать длинные строки на части и располагать эти части на отдельных строчках.

Фактически строковая константа — это массив символов. Во внутреннем представлении строки в конце обязательно присутствует нулевой символ '\0' , поэтому памяти для строки требуется на один байт больше, чем число символов, расположенных между двойными кавычками. Это означает, что на длину задаваемой строки нет ограничения, но чтобы определить ее длину, требуется просмотреть всю строку. Функция **strlen(s)** вычисляет длину строки s без учета завершающего ее символа '\0' . Ниже приводится наша версия этой функции:

```
/* strlen: возвращает длину строки s */
int strlen(char s[])
{
    int i;
    i = 0;
    while (s[i] != '\0')
        ++i;
    return i;
}
```

Функция **strlen** и некоторые другие, применяемые к строкам, описаны в стандартном заголовочном файле <**string.h**>.

Будьте внимательны и помните, что символьная константа и строка, содержащая один символ, не одно и то же: 'x' не то же самое, что "x". Запись 'x' обозначает целое значение, равное коду буквы x из стандартного символьного набора, а запись "x" - массив символов, который содержит один символ (букву x) и '\0'.

В Си имеется еще один вид константы - константа перечисления. Перечисление - это список целых констант, как, например, в

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

Первое имя в enum имеет значение 0, следующее - 1 и т.д. (если для значений констант не было явных спецификаций). Если не все значения специфицированы, то они продолжают прогрессию, начиная от последнего специфицированного значения, как в следующих двух примерах:

```
enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
                        NEWLINE = '\n', VTAB = '\v', RETURN = '\r'};
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
                        JUL, AUG, SEP, OCT, NOV, DEC};
                        /* FEB есть 2, MAR есть 3 и т.д. */
```

Имена в различных перечислениях должны отличаться друг от друга. Значения внутри одного перечисления могут совпадать.

Средство **enum** обеспечивает удобный способ присвоить константам имена, причем в отличие от **#define** значения констант при этом способе могут генерироваться автоматически. Хотя разрешается объявлять переменные типа **enum**, однако компилятор не обязан контролировать, входят ли присваиваемые этим переменным значения в их тип. Но сама возможность такой проверки часто делает enum лучше, чем #define. Кроме того, отладчик получает возможность печатать значения переменных типа enum в символьном виде.

2.4 Объявления

Все переменные должны быть объявлены раньше, чем будут использоваться, при этом некоторые объявления могут быть получены неявно - из контекста. Объявление специфицирует тип и содержит список из одной или нескольких переменных этого типа, как, например, в

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

Переменные можно распределять по объявлениям произвольным образом, так что указанные выше списки можно записать и в следующем виде:

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

Последняя форма записи занимает больше места, тем не менее она лучше, поскольку позволяет добавлять к каждому объявлению комментарий. Кроме того, она более удобна для последующих модификаций.

В своем объявлении переменная может быть инициализирована, как, например:

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

Инициализация неавтоматической переменной осуществляется только один раз - перед тем, как программа начнет выполняться, при этом начальное значение должно быть константным выражением. Явно инициализируемая автоматическая переменная получает начальное значение каждый раз при входе в функцию или блок, ее начальным значением может быть любое выражение. Внешние и статические переменные по умолчанию получают нулевые значения. Автоматические переменные, явным образом не инициализированные, содержат неопределенные значения ("мусор”).

К любой переменной в объявлении может быть применен квалификатор **const** для указания того, что ее значение далее не будет изменяться.

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

Применительно к массиву квалификатор **const** указывает на то, что ни один из его элементов не будет меняться. Указание **const** можно также применять к аргументу- массиву, чтобы сообщить, что функция не изменяет этот массив:

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

Реакция на попытку изменить переменную, помеченную квалификатором **const** зависит от реализации компилятора.

2.5 Арифметические операторы

Бинарными (т. е. с двумя операндами) арифметическими операторами являются +, -, *, /, а также

оператор деления по модулю **%**. Деление целых сопровождается отбрасыванием дробной части, какой бы она ни была. Выражение

```
x % y
```

дает остаток от деления x на y и, следовательно, нуль, если x делится на y нацело. Например, год является високосным, если он делится на 4, но не делится на 100. Кроме того, год является високосным, если он делится на 400. Следовательно,

```
if ((year % 4 == 0 && year % 100 !=0 || year % 400 == 0)
    printf("%d високосный год\n", year);
else
    printf("%d невисокосный год\n", year);
```

Оператор **%** к операндам типов **float** и **double** не применяется. В какую сторону (в сторону увеличения или уменьшения числа) будет усечена дробная часть при выполнении / и каким будет знак результата операции **%** с отрицательными операндами, зависит от машины.

Бинарные операторы + и **-** имеют одинаковый приоритет, который ниже приоритета операторов *****, / и **%**, который в свою очередь ниже приоритета унарных операторов + и **-**. Арифметические операции одного приоритетного уровня выполняются слева направо.

В конце этой главы (параграф 2.12) приводится таблица 2.1,в которой представлены приоритеты всех операторов и очередность их выполнения.

2.6 Операторы отношения и логические операторы

Операторами отношения являются

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

Все они имеют одинаковый приоритет. Сразу за ними идет приоритет операторов сравнения на равенство:

```
==
!=
```

Операторы отношения имеют более низкий приоритет, чем арифметические, поэтому выражение вроде i < lim-1 будет выполняться так же, как i < (lim-1), т.е. как мы и ожидаем.

Более интересны логические операторы **&&** и ||. Выражения, между которыми стоят операторы && или ||, вычисляются слева направо. Вычисление прекращается, как только становится известна истинность или ложность результата. Многие Си- программы опираются на это свойство, как, например, цикл из функции getline, которую мы приводили в главе 1:

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

Прежде чем читать очередной символ, нужно проверить, есть ли для него место в массиве s, иначе говоря, сначала необходимо проверить соблюдение условия i < lim-1. Если это условие не выполняется, мы не должны продолжать вычисление, в частности читать следующий символ. Так же было бы неправильным сравнивать c и EOF до обращения к getchar; следовательно, и вызов getchar, и присваивание должны выполняться перед указанной проверкой.

Приоритет оператора **&&** выше, чем таковой оператора ||, однако их приоритеты ниже, чем приоритет операторов отношения и равенства. Из сказанного следует, что выражение вида

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

не нуждается в дополнительных скобках. Но, так как приоритет **!=** выше, чем приоритет присваивания, в

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

скобки необходимы, чтобы сначала выполнить присваивание, а затем сравнение с '\n'.

По определению численным результатом вычисления выражения отношения или логического выражения является 1, если оно истинно, и 0, если оно ложно.

Унарный оператор **!** преобразует ненулевой операнд в 0, а нуль в 1. Обычно оператор **!** используют в конструкциях вида

```
if (!valid)
```

что эквивалентно

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

Трудно сказать, какая из форм записи лучше. Конструкция вида !valid хорошо читается ("если не правильно”), но в более сложных выражениях может оказаться, что ее не так-то легко понять.

Упражнение 2.2. Напишите цикл, эквивалентный приведенному выше or- циклу, не пользуясь операторами && и ||.

2.7 Преобразования типов

Если операнды оператора принадлежат к разным типам, то они приводятся к некоторому общему типу. Приведение выполняется в соответствии с небольшим числом правил. Обычно автоматически производятся лишь те преобразования, которые без какой-либо потери информации превращают операнды с меньшим диапазоном значений в операнды с большим диапазоном, как, например, преобразование целого в число с плавающей точкой в выражении вроде f+i. Выражения, не имеющие смысла, например число с плавающей точкой в роли индекса, не допускаются. Выражения, в которых могла бы теряться информация (скажем, при присваивании длинных целых переменным более коротких типов или при присваивании значений с плавающей точкой целым переменным), могут повлечь за собой предупреждение, но они допустимы.

Значения типа **char** - это просто малые целые, и их можно свободно использовать в арифметических выражениях, что значительно облегчает всевозможные манипуляции с символами. В качестве примера приведем простенькую реализацию функции atoi, преобразующей последовательность цифр в ее числовой эквивалент.

```
/* atoi: преобразование s в целое */
int atoi(char s[])
{
    int i, n;
    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9';++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

Как мы уже говорили в главе 1, выражение

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

дает числовое значение символа, хранящегося в s[i], так как значения '0', '1' и пр. образуют непрерывную возрастающую последовательность.

Другой пример приведения **char** к **int** связан с функцией lower, которая одиночный символ из набора ASCII, если он является заглавной буквой, превращает в строчную. Если же символ не является заглавной буквой, lower его не изменяет.

```
/* lower: преобразование c в строчную, только для ASCII */
int lower(int c)
{
    if (c >= 'A' && c <='Z')
        return c +'a'-'A';
    else
        return c;
}
```

В случае ASCII эта программа будет работать правильно, потому что между одноименными буквами верхнего и нижнего регистров - одинаковое расстояние (если их рассматривать как числовые значения). Кроме того, латинский алфавит - плотный, т. е. между буквами A и Z расположены только буквы. Для набора EBCDIC последнее условие не выполняется, и поэтому наша программа в этом случае будет преобразовывать не только буквы.

Стандартный заголовочный файл <**ctype.h**>, описанный в приложении B, определяет семейство функций, которые позволяют проверять и преобразовывать символы независимо от символьного набора. Например, функция tolower(c) возвращает букву c в коде нижнего регистра, если она была в коде верхнего регистра, поэтому tolower - универсальная замена функции lower, рассмотренной выше. Аналогично проверку

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

можно заменить на isdigit(c)

Далее мы будем пользоваться функциями из <ctype.h>.

Существует одна тонкость, касающаяся преобразования символов в целые числа: язык не определяет, являются ли переменные типа **char** знаковыми или беззнаковыми. При преобразовании **char** в **int** может ли когда- нибудь получиться отрицательное целое? На машинах с разной архитектурой ответы могут отличаться. На некоторых машинах значение типа **char** с единичным старшим битом будет превращено в отрицательное целое (посредством "распространения знака”). На других - преобразование **char** в **int** осуществляется добавлением нулей слева, и, таким образом, получаемое значение всегда положительно.

Гарантируется, что любой символ из стандартного набора печатаемых символов никогда не будет отрицательным числом, поэтому в выражениях такие символы всегда являются положительными операндами. Но произвольный восьмибитовый код в переменной типа **char** на одних машинах может быть отрицательным числом, а на других - положительным. Для совместимости переменные типа **char**, в которых хранятся несимвольные данные, следует специфицировать явно как **signed** или **unsigned**.

Отношения вроде i > j и логические выражения, перемежаемые операторами **&&** и ||, определяют выражение-условие, которое имеет значение 1, если оно истинно, и 0, если ложно. Так, присваивание

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

установит d в значение 1, если c есть цифра, и 0 в противном случае. Однако функции, подобные isdigit, в качестве истины могут выдавать любое ненулевое значение. В местах проверок внутри if, while, for и пр. "истина” просто означает "не нуль”.

Неявные арифметические преобразования, как правило, осуществляются естественным образом. В общем случае, когда оператор вроде + или * с двумя операндами (бинарный оператор) имеет разнотипные операнды, прежде чем операция начнет выполняться, "низший” тип повышается до "высшего”. Результат будет иметь высший тип. В параграфе 6 приложения A правила преобразования сформулированы точно. Если же в выражении нет беззнаковых операндов, можно удовлетвориться следующим набором неформальных правил:

- Если какой-либо из операндов принадлежит типу **long double**, то и другой приводится к **long double**.
- В противном случае, если какой-либо из операндов принадлежит типу **double**, то и другой приводится к **double**.
- В противном случае, если какой-либо из операндов принадлежит типу **float**, то и другой приводится к **float**.
- В противном случае операнды типов **char** и **short** приводятся к **int**.
- И наконец, если один из операндов типа **long**, то и другой приводится к **long**.

Заметим, что операнды типа **float** не приводятся автоматически к типу **double**; в этом данная версия языка отличается от первоначальной. Вообще говоря, математические функции, аналогичные собранным в библиотеке <**math.h**>, базируются на вычислениях с двойной точностью. В основном **float**

используется для экономии памяти на больших массивах и не так часто - для ускорения счета на тех машинах, где арифметика с двойной точностью слишком дорога с точки зрения расхода времени и памяти.

Правила преобразования усложняются с появлением операндов типа **unsigned**. Проблема в том, что сравнения знаковых и беззнаковых значений зависят от размеров целочисленных типов, которые на разных машинах могут отличаться. Предположим, что значение типа **int** занимает 16 битов, а значение типа **long** - 32 бита. Тогда -1L < 1U, поскольку 1U принадлежит типу **unsigned int** и повышается до типа **signed long**. Но –1L >1UL, так как -1L повышается до типа **unsigned long** и воспринимается как большое положительное число.

Преобразования имеют место и при присвоениях: значение правой части присвоения приводится к типу левой части, который и является типом результата.

Тип **char** превращается в **int** путем распространения знака или другим описанным выше способом.

Тип **long int** преобразуются в **short int** или в значения типа **char** путем отбрасывания старших разрядов. Так, в

```
int i;
char c;
i = c;
c = i;
```

значение c не изменится. Это справедливо независимо от того, распространяется знак при переводе **char** в **int** или нет. Однако, если изменить очередность присваиваний, возможна потеря информации.

Если x принадлежит типу **float**, а i - типу **int**, то и x=i, и i=z вызовут преобразования, причем перевод **float** в **int** сопровождается отбрасыванием дробной части. Если **double** переводится во **float**, то значение либо округляется, либо обрезается; это зависит от реализации.

Так как аргумент в вызове функции есть выражение, при передаче его функции также возможно преобразование типа. При отсутствии прототипа (функции аргументы тина **char** и **short** переводятся в **int**, а **float** - в **double**. Вот почему мы объявляли аргументы типа **int** или **double** даже тогда, когда в вызове функции использовали аргументы типа **char** или **float**.

И наконец, для любого выражения можно явно ("насильно”) указать преобразование его типа, используя унарный оператор, называемый приведением. Конструкция вида

```
(имя-типа) выражение
```

приводит выражение к указанному в скобках типу по перечисленным выше правилам. Смысл операции приведения можно представить себе так: выражение как бы присваивается некоторой переменной указанного типа, и эта переменная используется вместо всей конструкции. Например, библиотечная функция sqrt рассчитана на аргумент типа **double** и выдает чепуху, если ей подсунуть что-нибудь другое (sqrt описана в). Поэтому, если n имеет целочисленный тип, мы можем написать

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

и перед тем, как значение n будет передано функции, оно будет переведено в **double**. Заметим, что операция приведения всего лишь вырабатывает значение n указанного типа, но саму переменную n не затрагивает. Приоритет оператора приведения столь же высок, как и любого унарного оператора, что зафиксировано в таблице, помещенной в конце этой главы.

В том случае, когда аргументы описаны в прототипе функции, как тому и следует быть, при вызове функции нужное преобразование выполняется автоматически. Так, при наличии прототипа функции sqrt:

```
double sqrt(double);
```

перед обращением к sqrt в присваивании

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

целое 2 будет переведено в значение **double** 2.0 автоматически без явного указания операции приведения.

Операцию приведения проиллюстрируем на переносимой версии генератора псевдослучайных чисел и функции, инициализирующей "семя". И генератор, и функция входят в стандартную библиотеку.

```
unsigned long int next = 1;
/* rand: возвращает псевдослучайное целое 0...32767 */
int rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: устанавливает "семя" для rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Упражнение 2.3. Напишите функцию htol(s), которая преобразует последовательность шестнадцатеричных цифр, начинающуюся с 0x или 0X, в соответствующее целое. Шестнадцатеричными цифрами являются символы 0...9, a...f, A...F.

2.8 Операторы инкремента и декремента

В Си есть два необычных оператора, предназначенных для увеличения и уменьшения переменных. Оператор инкремента ++ добавляет 1 к своему операнду, а оператор декремента -- вычитает 1. Мы уже неоднократно использовали ++ для наращивания значения переменных, как, например, в

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

Необычность операторов ++ и -- в том, что их можно использовать и как префиксные (помещая перед переменной: ++n), и как постфиксные (помещая после переменной: n++) операторы. В обоих случаях значение n увеличивается на 1, но выражение ++n увеличивает n до того, как его значение будет использовано, а n++ - после того. Предположим, что n содержит 5, тогда

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

установит x в значение 5, а

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

установит x в значение 6. И в том и другом случае n станет равным 6. Операторы инкремента и декремента можно применять только к переменным. Выражения вроде (i+j)++ недопустимы.

Если требуется только увеличить или уменьшить значение переменной (но не получить ее значение), как например

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

то безразлично, какой оператор выбрать - префиксный или постфиксный. Но существуют ситуации, когда требуется оператор вполне определенного типа. Например, рассмотрим функцию squeeze(s, c), которая удаляет из строки s все символы, совпадающие с c:

```
/* squeeze: удаляет все c из s*/
void squeeze(char s[], int c)
{
    int i, j;
    for (i = j =0; s[i] != '\0'; i++)
        if (s[i] != c)
            s[j++] = s[i];
        s[i] = '\0';
}
```

Каждый раз, когда встречается символ, отличный от c, он копируется в текущую j-ю позицию, и только после этого переменная j увеличивается на 1, подготавливаясь таким образом к приему следующего символа. Это в точности совпадает со следующими действиями:

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

Другой пример - функция getline, которая нам известна по главе 1. Приведенную там запись

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

можно переписать более компактно:

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

В качестве третьего примера рассмотрим стандартную функцию strcat(s,t), которая строку t помещает в конец строки s. Предполагается, что в s достаточно места, чтобы разместить там суммарную строку. Мы написали strcat так, что она не возвращает никакого результата. На самом деле библиотечная strcat возвращает указатель на результирующую строку.

```
/* strcat: помещает t в конец s; s достаточно велика */
void strcat (char s[], char t[])
{
    int i, j;
    i = j = 0;
    while (s[i] != '\0') /* находим конец s */
        i++;
    while ((s[i++] = t[j++]) != '\0') /* копируем t */
        ;
}
```

При копировании очередного символа из t в s постфиксный оператор ++ применяется и к i, и к j, чтобы на каждом шаге цикла переменные i и j правильно отслеживали позиции перемещаемого символа.

Упражнение 2.4. Напишите версию функции squeeze(s1,s2), которая удаляет из s1 все символы, встречающиеся в строке s2.

Упражнение 2.5. Напишите функцию any(s1,s2), которая возвращает либо ту позицию в s1, где стоит первый символ, совпавший с любым из символов в s2, либо -1 (если ни один символ из s1 не совпадает с символами из s2). (Стандартная библиотечная функция strpbrk делает то же самое, но выдает не номер позиции символа, а указатель на символ.)

2.9 Побитовые операторы

В Си имеются шесть операторов для манипулирования с битами. Их можно применять только к целочисленным операндам, т. е. к операндам типов **char**, **short**, **int** и long, знаковым и беззнаковым.

```
& - побитовое И
| - побитовое ИЛИ
^ - побитовое исключающее ИЛИ.
<< - сдвиг влево.
>> - сдвиг вправо.
~ - побитовое отрицание (унарный).
```

Оператор **&** (побитовое И) часто используется для обнуления некоторой группы разрядов. Например

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

обнуляет в n все разряды, кроме младших семи.

Оператор | (побитовое ИЛИ) применяют для установки разрядов; так,

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

устанавливает единицы в тех разрядах x, которым соответствуют единицы в SET_ON.

Оператор ^ (побитовое исключающее ИЛИ) в каждом разряде установит 1, если соответствующие разряды операндов имеют различные значения, и 0, когда они совпадают.

Поразрядные операторы **&** и | следует отличать от логических операторов **&&** и ||, которые при вычислении слева направо дают значение истинности. Например, если x равно 1, а y равно 2, то x & y даст нуль, а x && y - единицу.

Операторы << и >> сдвигают влево или вправо свой левый операнд на число битовых позиций, задаваемое правым операндом, который должен быть неотрицательным. Так, x << 2 сдвигает значение x влево на 2 позиции, заполняя освобождающиеся биты нулями, что эквивалентно умножению x на 4. Сдвиг вправо беззнаковой величины всегда сопровождается заполнением освобождающихся разрядов нулями. Сдвиг вправо знаковой величины на одних машинах происходит с распространением знака ("арифметический сдвиг”), на других - с заполнением освобождающихся разрядов нулями ("логический сдвиг”).

Унарный оператор ~ поразрядно "обращает” целое т. е. превращает каждый единичный бит в нулевой и наоборот. Например

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

обнуляет в x последние 6 разрядов. Заметим, что запись x & ~077 не зависит от длины слова, и, следовательно, она лучше, чем x & 0177700, поскольку последняя подразумевает, что x занимает 16 битов. Не зависимая от машины форма записи ~077 не потребует дополнительных затрат при счете, так как ~077 - константное выражение, которое будет вычислено во время компиляции.

Для иллюстрации некоторых побитовых операций рассмотрим функцию getbits(x, p, n), которая формирует поле в n битов, вырезанных из x, начиная с позиции p, прижимая его к правому краю. Предполагается, что 0-й бит - крайний правый бит, а n и p- осмысленные положительные числа. Например, getbits(x,4,3) вернет в качестве результата 4, 3 и 2-й биты значения x, прижимая их к правому краю. Вот эта функция:

```
/* getbits: получает n бит, начиная с p-й позиции */
unsigned getbits(unsigned x, int p, int n)
{
    return (x >> (p+1-n)) & ~(~0 << n);
}
```

Выражение x >> (p+1-n) сдвигает нужное нам поле к правому краю. Константа ~0 состоит из одних единиц, и ее сдвиг влево на n бит (~0 << n) приведет к тому, что правый край этой константы займут n нулевых разрядов. Еще одна операция побитовой инверсии ~ позволяет получить справа n единиц.

Упражнение 2.6. Напишите функцию setbits(x, p, n, y), возвращающую значение x, в котором n битов, начиная с p-й позиции, заменены на n правых разрядов из y (остальные биты не изменяются).

Упражнение 2.7. Напишите функцию invert(x, p, n), возвращающую значение x с инвертированными n битами, начиная с позиции p (остальные биты не изменяются).

Упражнение 2.8. Напишите функцию rightrot (x, n), которая циклически сдвигает x вправо на n разрядов.

2.10 Операторы и выражения присваивания

Выражение

```
i = i + 2;
```

в котором стоящая слева переменная повторяется и справа, можно написать в сжатом виде:

```
i += 2;
```

Оператор +=, как и =, называется оператором присваивания.

Большинству бинарных операторов (аналогичных + и имеющих левый и правый операнды) соответствуют операторы присваивания **op**=, где op - один из операторов

```
+
-
```

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

Если *выр$_1$* и *выр$_2$* - выражения, то

выр$_1$ op= выр$_2$

Эквивалентно

выр$_1$ = (выр$_1$) op (выр$_2$)

с той лишь разницей, что выр$_1$ вычисляется только один раз. Обратите внимание на скобки вокруг выр$_2$:

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

эквивалентно

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

но не

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

В качестве примера приведем функцию bitcount, подсчитывающую число единичных битов в своем аргументе целочисленного типа.

```
/* bitcount: подсчет единиц в x */
int bitcount(unsigned x)
{
    int b;
    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            b++;
    return b;
}
```

Независимо от машины, на которой будет работать эта программа, объявление аргумента x как unsigned гарантирует, что при правом сдвиге освобождающиеся биты будут заполняться нулями, а не знаковым битом.

Помимо краткости операторы присваивания обладают тем преимуществом, что они более соответствуют тому, как человек мыслит. Мы говорим "прибавить 2 к i" или "увеличить i на 2", а не "взять i, добавить 2 и затем вернуть результат в i", так что выражение i+=2 лучше, чем i=i+2. Кроме того, в сложных выражениях вроде

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

благодаря оператору присваивания += запись становится более легкой для понимания, так как читателю при такой записи не потребуется старательно сравнивать два длинных выражения, совпадают ли они, или выяснять, почему они не совпадают. Следует иметь в виду и то, что подобные операторы присваивания могут помочь компилятору сгенерировать более эффективный код.

Мы уже видели, что присваивание вырабатывает значение и может применяться внутри выражения: вот самый расхожий пример:

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

В выражениях встречаются и другие операторы присваивания (+=, -= и т. д.), хотя и реже. Типом и значением любого выражения присваивания являются тип и значение его левого операнда после завершения присваивания.

Упражнение 2.9. Применительно к числам, в представлении которых использован дополнительный код, выражение x &= (x-1) уничтожает самую правую 1 в x. Объясните, почему. Используйте это

наблюдение при написании более быстрого варианта функции bitcount.

2.11 Условные выражения

Инструкции

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

пересылают в z большее из двух значений a и b. Условное выражение, написанное с помощью тернарного (т. е. имеющего три операнда) оператора "? : ", представляет собой другой способ записи этой и подобных ей конструкций. В выражении

```
выр1 ? выр2 : выр3
```

первым вычисляется выражение $выр_1$. Если его значение не нуль (истина), то вычисляется выражение $выр_2$, и значение этого выражения становится значением всего условного выражения. В противном случае вычисляется выражение $выр_3$ и его значение становится значением условного выражения. Следует отметить, что из выражений $выр_2$ и $выр_3$ вычисляется только одно из них. Таким образом, чтобы установить в z большее из a и b, можно написать

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

Следует заметить, что условное выражение и в самом деле является выражением, и его можно использовать в любом месте, где допускается выражение. Если $выр_2$ и $выр_3$ принадлежат разным типам, то тип результата определяется правилами преобразования, о которых шла речь в этой главе ранее. Например, если f имеет тип **float**, а n - тип **int**, то типом выражения

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

будет **float** вне зависимости от того, положительно значение n или нет.

Заключать в скобки первое выражение в условном выражении не обязательно, так как приоритет **?:** очень низкий (более низкий приоритет имеет только присваивание), однако мы рекомендуем всегда это делать, поскольку благодаря обрамляющим скобкам условие в выражении лучше воспринимается.

Условное выражение часто позволяет сократить программу. В качестве примера приведем цикл, обеспечивающий печать n элементов массива по 10 на каждой строке с одним пробелом между колонками; каждая строка цикла, включая последнюю, заканчивается символом новой строки:

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

Символ новой строки посылается после каждого десятого и после n-го элемента. За всеми другими элементами следует пробел. Эта программа выглядит довольно замысловато, зато она более компактна, чем эквивалентная программа с использованием if-else. Вот еще один хороший пример :

```
printf("Вы имеете %d элемент%s: \n", n, (n%10 == 1 && n%100 != 11) ?
    " " : ((n%100 < 10 || n%100 > 20) && n%10 >= 2 && n%10 <= 4) ?
        "а" : "ов");
```

Упражнение 2.10. Напишите функцию lower, которая переводит большие буквы в малые, используя условное выражение (а не конструкцию if-else).

2.12 Приоритет и очередность вычислений

В таблице 2.1 показаны приоритеты и очередность вычислений всех операторов, включая и те, которые мы еще не рассматривали. Операторы, перечисленные на одной строке, имеют одинаковый приоритет: строки упорядочены по убыванию приоритетов; так, например, *, / и % имеют одинаковый приоритет, который выше, чем приоритет бинарных + и -. "Оператор” () относится к вызову функции. Операторы -> и . (точка) обеспечивают доступ к элементам структур; о них пойдет речь в главе 6, там же будет

рассмотрен и оператор sizeof (размер объекта). Операторы * (косвенное обращение по указателю) и & (получение адреса объекта) обсуждаются в главе 5. Оператор "запятая" будет рассмотрен в главе 3.

Таблица 2.1. Приоритеты и очередность вычислений операторов

Операторы											**Выполняются**
()	**[]**	**->**	**.**								слева направо
!	**~**	**++**	**--**	**+**	**-**	*****	**&**	**(type)**	**sizeof**		справа налево
*****	**/**	**%**									слева направо
+	**-**										слева направо
<<	**>>**										слева направо
<	**<=**	**>**	**>=**								слева направо
==	**!=**										слева направо
&											слева направо
^											слева направо
|											слева направо
&&											слева направо
||											слева направо
?:											справа налево
=	**+=**	**-=**	***=**	**/=**	**%=**	**&=**	**^=**	**|=**	**<<=**	**>>=**	справа налево
,											слева направо

Примечание. Унарные операторы **+**, **-**, ***** и **&** имеют более высокий приоритет, чем те же бинарные операторы.

Заметим, что приоритеты побитовых операторов **&**, ^ и | ниже, чем приоритет == и **!=** , из-за чего в побитовых проверках, таких как

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

чтобы получить правильный результат, приходится использовать скобки. Си подобно многим языкам не фиксирует очередность вычисления операндов оператора (за исключением **&&**, ||, **?:** и **,**). Например, в инструкции вида

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

f может быть вычислена раньше g или наоборот. Из этого следует, что если одна из функций изменяет значение переменной, от которой зависит другая функция, то помещаемый в x результат может зависеть от очередности вычислений. Чтобы обеспечить нужную последовательность вычислений, промежуточные результаты можно запоминать во временных переменных.

Очередность вычисления аргументов функции также не определена, поэтому на разных компиляторах

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

может давать несовпадающие результаты. Результат вызова функции зависит от того, когда компилятор сгенерирует команды увеличения n - до или после обращения к power. Чтобы обезопасить себя от возможного побочного эффекта, достаточно написать

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

Обращения к функциям, вложенные присвоения, инкрементные и декрементные операторы дают

"побочный эффект", проявляющийся в том, что при вычислении выражения значения некоторых переменных изменяются. В любом выражении с побочным эффектом может быть скрыта трудно просматриваемая зависимость результата выражения от очередности изменения значений переменных, входящих в выражение. В такой, например, типично неприятной ситуации

```
a[i] = i++; /* I.B.: doubtful example */
```

возникает вопрос: массив a индексируется старым или измененным значением i? Компиляторы могут по-разному генерировать программу, что проявится в интерпретации данной записи. Стандарт сознательно устроен так, что большинство подобных вопросов оставлено на усмотрение компиляторов, так как лучший порядок вычислений определяется архитектурой машины. Стандартом только гарантируется, что все побочные эффекты при вычислении аргументов проявятся перед входом в функцию. Правда, в примере с printf это нам не поможет.

Мораль такова: писать программы, зависящие от очередности вычислений, - плохая практика, какой бы язык вы ни использовали. Естественно, надо знать, чего следует избегать, но если вы не знаете, как образуются побочные эффекты на разных машинах, то лучше и не рассчитывать выиграть на особенностях частной реализации.

[Назад] [Содержание] [Вперед]

[Главная] [Гостевая]