Оглавление

7. Операции (продолжение)

Преобразования типов

При участии в арифметических операциях операндов разных типов, перед выполнением операции осуществляется преобразование типов так, чтобы не было потери точности. Затем осуществляется сама операция.

Типы данных в порядке возрастания точности:

```
char, short, int, long, float, double.
```

Пример:

```
int a, c;
float d, e;
e = d*(c+e);  /* c будет преобразовано в float */
a = 'A' + 20; /* 'A' будет преобразовано в int */
```

В некоторых случаях правила преобразования типов не позволяют получить верный результат, например:

```
int a, b;
float c;
a = 7;
b = 14;
c = a/b;        /* результат 0 */
```

Для любого выражения можно явно 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.

Для нашего примера применение операции преобразования типа будет выглядеть так:

```
c = (float)a / (float)b;
```

Операция преобразования типа имеет приоритет более высокий, чем арифметическая операция.

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.

Операции 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.

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 имеет значение 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.

Операторы и выражения присваивания

Выражение

```
i = i + 2
```

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

```
i += 2
```

Операция +=, как и =, называется *операцией присваивания*.

Большинству бинарных операторов (аналогичных + и имеющих левый и правый операнды) соответствуют операции 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.

Использование операций ++, --, +=, -=, *= , /=, %= вместо обычных не является обязательным, но их применение считается хорошим стилем программирования на языке Си.

Битовые операции

Любые данные, записанные в память ЭВМ, как известно, представляют собой последовательность бит, т.е. последовательность нулей и единиц. Например, любое число типа int будет занимать 2 байта в памяти, т.е 16 бит. Его можно рассматривать двояко: либо как целое число (так и делается при выполнении операций *, /, +, - , %), либо как последовательность бит, что возможно при использовании битовых операций.

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.

В Си имеются следующие *побитовые* операции, которые аналогичны логическим операциям, но выполняются независимо над каждым битом данных. Если операция двуместная, то она выполняется над соответствующими битами операндов.

~ побитовое отрицание (унарная),

& побитовое "и" (бинарная),

^ побитовое "исключающие или" (бинарная),

| побитовое "или" (бинарная).

Результаты побитовых операций, a и b — отдельные биты чисел:

a	b	~a	a & b	a ^ b	a | b
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	0	1

Рассмотрим несколько примеров.

Операция & (побитовое И) часто используется для обнуления некоторой группы разрядов. Например,

```
n = n & 65280;
```

Обозначим двоичное (битовое) представление значения n до выполнения операции через $a_{15}a_{14}...a_1a_0$. В таблице показаны двоичные (битовые) представления n, 65280, n&65280.

n	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
65280	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
n&65280	a_{15}	a_{14}	a_{13}	a_{12}	a_{11}	a_{10}	a_9	a_8	0	0	0	0	0	0	0	0

Мы видим, что оператор `n = n & 65280;` обнуляет те разряды n, в которых в двоичном представлении 65280 находятся нули, т. е. последние 8 разрядов n.

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

x = x | SET_ON;

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

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

Унарная операция ~ поразрядно «обращает» целое, т. е. превращает каждый единичный бит в нулевой и наоборот.

К битовым операциям относятся операции сдвига << и >> :

`a << b` сдвиг битов переменной a влево на b позиций,

`a >> b` сдвиг битов переменной a вправо на b позиций.

Так, x << 3 сдвигает значение x влево на 3 позиции, заполняя освобождающиеся биты нулями, что эквивалентно умножению x на 8.

Выражение x >> 2 сдвигает значение x вправо на 2 позиции, заполняя освобождающиеся биты нулями, что эквивалентно делению нацело x на 4.

Сдвиг вправо беззнаковой величины всегда сопровождается заполнением освобождающихся разрядов нулями. Сдвиг вправо знаковой величины осуществляется разными компиляторами по-разному.

Все битовые операции выполняются слева направо. В следующей строке приведены битовые операции в порядке уменьшения их приоритета.

`~, << >>, &, ^, |`

Для двуместных битовых операций определены дополнительные операции присваивания:

a <<= b; эквивалентно a = a << b,

a >>= b; эквивалентно a = a >> b,

a &= b; эквивалентно a = a & b,

a ^= b; эквивалентно a = a ^ b,

a |= b; эквивалентно a = a | b.

Приоритеты операций

Приоритеты и направление рассмотренных выше операций сведены в следующую таблицу. Операции одинакового приоритета объединены в группы, чем выше положение группы в таблице, тем выше приоритет операций группы.

В таблице показаны приоритеты операций, в том числе и тех, про которые еще не говорилось.

Таблица приоритетов операций

Операции одного приоритета	Направление выполнения операции
`! ~ ++ -- (тип) sizeof`	←
`* / %`	→
`+ - (бинарные)`	→
`<< >>`	→
`< <= > >=`	→
`== !=`	→
`&`	→
`^`	→
`|`	→
`&&`	→
`||`	→
`= *= /= %= += -= <<= >>= &= ^= |=`	←

Операции, перечисленные на одной строке, имеют одинаковый приоритет; строки упорядочены по убыванию приоритетов; так, например, *, / и % имеют одинаковый приоритет, который выше, чем приоритет бинарных + и -. Унарные операции +, - имеют более высокий приоритет, чем те же бинарные операции.

8. Управление

Порядок, в котором выполняются вычисления, определяется *инструкциями управления*.

Выражения и блоки

Как было уже описано выше, выражение, скажем `x = 0, i++` или `printf (...)` , становится оператором, если в конце его поставить точку с запятой, например:

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

В Си точка с запятой является заключающим символом оператора, а не разделителем, как в языке Паскаль.

Фигурные скобки { и } используются для объединения объявлений и инструкций в *составной оператор*, или *блок*, чтобы с точки зрения синтаксиса эта новая конструкция воспринималась как одна инструкция.

Фигурные скобки, обрамляющие группу операторов, образующих тело функции, — это один пример; второй пример — это скобки, объединяющие операторы, помещенные после if, else, while или for.

После правой закрывающей фигурной скобки в конце блока точка с запятой не ставится.

Конструкция if-else

Конструкция if-else используется для принятия решения. Формально ее синтаксисом является:

```
if (выражение)
оператор 1
else
оператор 2
```

причем else-часть может и отсутствовать.

Если в качестве оператора 1 или оператора 2 используется группа операторов, то ее записывают как составной оператор, заключая ее в операторные скобки "{" и "}". Сначала вычисляется выражение, и, если оно истинно (т. е. отлично от нуля), выполняется оператор1. Если выражение ложно (т. е. его значение равно нулю) и существует else-часть, то выполняется оператор2.

Так как if просто проверяет числовое значение выражения, условие иногда можно записывать в сокращенном виде. Так, запись

```
if (выражение)
```

короче, чем

```
if (выражение ! = 0)
```

Иногда такие сокращения естественны и ясны, в других случаях, наоборот, затрудняют понимание программы.

При наличии вложенных операторов if следует иметь в виду, что ключевое слово else всегда относится к ближайшему if, как показано в следующем примере:

```
if ( b > a )
    if ( c < d ) k= sin(x);
        else t = cos(x);
    else c = tan( x );
```

Для того чтобы избежать ошибок рекомендуется использовать скобки, указывающие границы блока.

```
if ( b < a )
    {
        if( c < d ) k = sin( x );
                    else t = cos( x );
    }
    else c = tan( x );
```

Особенно это важно если у вложенного оператора if отсутствует else, например:

```
if( b > a )
    {
        if( c < d ) k = sin( x );
    }
```

```
    else
        c = tan( x );
```

В некоторых случаях вместо оператора if удобно использовать условную операцию ? :, которая позволяет сократить запись программ и число используемых переменных.

```
результат = выражение_0 ? выражение_1: выражение_2
```

Результат условной операции равен выражению_1, если выражение_0 не равно 0 и выражению_2 в противном случае.

Например, следующий оператор

```
if ( x>a ) f = sin( x-a ); else f = sin(x);
```

можно заменить условной операцией, и сразу вычислить f:

```
f = sin( x>a ? x-a : x );
```

Очевидно, в последнем случае получился более короткий код, поскольку обращений к переменной f и функции sin вдвое меньше.

Конструкция else-if

Конструкция

```
if (выражение)
    оператор
    else if (выражение)
            оператор
```

```
else if (выражение)
    оператор
    else if (выражение)
        оператор
        else
        оператор
```

встречается так часто, что о ней стоит поговорить особо. Приведенная последовательность операторов if — самый общий способ описания многоступенчатого принятия решения. Выражения вычисляются по порядку; как только встречается выражение со значением "истина", выполняется соответствующий ему оператор; на этом последовательность проверок завершается. Оператор может быть и составным. Последняя else-часть срабатывает, если не выполняются все предыдущие условия. Иногда в последней части не требуется производить никаких действий, в этом случае фрагмент

```
else
инструкция
```

можно опустить или использовать для фиксации ошибочной ("невозможной") ситуации.

Переключатель switch

Инструкция switch используется для выбора одного из многих путей. Она проверяет, совпадает ли значение выражения с одним из значений, входящих в некоторое множество целых констант, и выполняет

соответствующую этому значению ветвь программы:

```
switch (выражение) {
case конст-выр: инструкции
case конст-выр: инструкции
default: инструкции
}
```

Каждая ветвь case помечена одной или несколькими целочисленными константами или же константными выражениями. Вычисления начинаются с той ветви case, в которой константа совпадает со значением выражения. Константы всех ветвей case должны отличаться друг от друга. Если выяснилось, что ни одна из констант не подходит, то выполняется ветвь, помеченная словом default, если таковая имеется, в противном случае ничего не делается. Ветви case и default можно располагать в любом порядке.

Инструкция break вызывает немедленный выход из переключателя switch. Поскольку выбор ветви case реализуется как переход на метку, то после выполнения одной ветви case, если ничего не предпринять, программа провалится вниз на следующую ветвь. Инструкции break и return — наиболее распространенные средства выхода из переключателя.

"Сквозное" выполнение ветвей case вызывает смешанные чувства. С одной стороны, это хорошо, поскольку позволяет несколько ветвей case объединить в одну, как мы и поступили с цифрами в нашем примере. Но с другой это означает, что в конце почти каждой ветви придется ставить break, чтобы избежать перехода к следующей. Последовательный проход по ветвям — вещь ненадежная, это чревато ошибками, особенно

приизменении программы. За исключением случая с несколькими метками для одного вычисления, старайтесь по возможности реже пользоваться сквозным проходом, но если уж вы его применяете, обязательно комментируйте эти особые места.

Даже в конце последней ветви (после default в нашем примере) помещайте инструкцию break, хотя с точки зрения логики в ней нет никакой необходимости. Но эта маленькая предосторожность спасет вас, когда однажды вам потребуется добавить в конец еще одну ветвь case.

Циклы while и for

В цикле

```
while (выражение)
оператор
```

вычисляется выражение. Если его значение отлично от нуля, то выполняется оператор, и вычисление выражения повторяется. Этот цикл продолжается до тех пор, пока выражение не станет равным нулю, после чего оператор уже не выполнится и вычисления продолжатся с точки, расположенной сразу за инструкцией.

Инструкция for

```
for (выр 1; выр 2; выр 3 )
инструкция
```

эквивалентна конструкции

```
выр 1;
while (выр 2 ) {
инструкция
```

```
выр 3;
}
```

С точки зрения грамматики три компоненты цикла for представляют собой произвольные выражения, но чаще выр 1 и выр 3 — это присваивания или вызовы функций, а выр 2 — выражение отношения. Любое из этих

трех выражений может отсутствовать, но точку с запятой опускать нельзя. При отсутствии выр 1 или выр 3 считается, что их просто нет в конструкции цикла; при отсутствии выр 2 предполагается, что его значение как бы всегда истинно. Например,

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

есть "бесконечный" цикл, выполнение которого, вероятно, прерывается каким-то другим способом, например, с помощью инструкций break или return.

Там, где есть простая инициализация и пошаговое увеличение значения некоторой переменной, больше подходит цикл for, так как в этом цикле организующая его часть сосредоточена в начале записи. Например, начало цикла, обрабатывающего первые n элементов массива, имеет следующий вид:

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

В конструкции for бывает удобно использовать еще один оператор Си —

это "," (запятая), которую чаще всего используют в инструкции for. Пара выражений, разделенных запятой, вычисляется слева направо. Типом и значением результата являются тип и значение правого выражения, что позволяет в инструкции for в каждой из трех компонент иметь по нескольку выражений, например вести два индекса параллельно. Продемонстрируем это на примере

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

Запятыми как операторами следует пользоваться умеренно.

Цикл do-while

Как мы говорили в главе 1, в циклах while и for проверка условия окончания цикла выполняется наверху. В Си имеется еще один вид цикла, do-while, в котором эта проверка в отличие от while и for делается внизу

после каждого прохождения тела цикла, т. е. после того, как тело выполнится хотя бы один раз. Цикл do-while имеет следующий синтаксис:

```
do
инструкция
while (выражение);
```

Сначала выполняется инструкция, затем вычисляется выражение. Если оно истинно, то инструкция выполняется снова и т. д. Когда выражение становится ложным, цикл заканчивает работу. Цикл do-while

эквивалентен циклу repeat-until в Паскале с той лишь разницей, что в первом случае указывается условие продолжения цикла, а во втором — условие его окончания.

Опыт показывает, что цикл do-while используется гораздо реже, чем while и for . Тем не менее, потребность в нем время от времени возникает.

Инструкции break и continue

Иногда бывает удобно выйти из цикла не по результату проверки, осуществляемой в начале или в конце цикла, а каким-то другим способом. Такую возможность для циклов for, while и do-while, а также для переключателя switch предоставляет инструкция break. Эта инструкция вызывает немедленный выход из самого внутреннего из объемлющих ее циклов или переключателей.

Инструкция continue в чем-то похожа на break, но применяется гораздо реже. Она вынуждает ближайший объемлющий ее цикл (for, while или do-while) начать следующий шаг итерации. Для while и do-while это означает немедленный переход к проверке условия, а для for — к приращению шага. Инструкцию continue можно применять только к циклам, но не к switch. Внутри переключателя switch, расположенного в цикле, она вызовет переход к следующей итерации этого цикла.

Вот фрагмент программы, обрабатывающий только неотрицательные элементы массива a (отрицательные

пропускаются).

```
for ( i = 0 ; i < n; i++) {
if (a[i] < 0) /* пропуск отрицательных элементов */continue;
```

```
/* обработка положительных элементов */
}
```

К инструкции continue часто прибегают тогда, когда оставшаяся часть цикла сложна, а замена условия в нем на противоположное и введение еще одного уровня приводят к слишком большому числу уровней вложенности.

Инструкция goto и метки

В Си имеются порицаемая многими инструкция goto и метки для перехода на них. Строго говоря, в этой инструкции нет никакой необходимости, и на практике почти всегда легко без нее обойтись.

Однако существуют случаи, в которых goto может пригодиться. Наиболее типична ситуация, когда нужно прервать обработку в некоторой глубоко вложенной структуре и выйти сразу из двух или большего числа вложенных циклов. Инструкция break здесь не поможет, так как она обеспечит выход только из самого внутреннего цикла. В качестве примера рассмотрим следующую конструкцию:

```
for (...)
for (...) {
...
if (катастрофа)
goto ошибка;
}
...
ошибка: ликвидировать беспорядок
```

Такая организация программы удобна, если подпрограмма обработки ошибочной ситуации не тривиальна и ошибка может встретиться в нескольких местах.

Метка имеет вид обычного имени переменной, за которым следует двоеточие. На метку можно перейти с помощью goto из любого места данной функции, т. е. метка видима на протяжении всей функции.

За исключением редких случаев, подобных только что приведенным, программы с применением goto, как правило, труднее для понимания и сопровождения, чем программы, решающие те же задачи без goto. К goto следует прибегать крайне редко, если использовать эту инструкцию вообще.