Книга по работе с WinAVR и AVR Studio

Роман Абраш
г. Новочеркасск
E-mail: **arv@radioliga.com**

Продолжение. Начало в №1/2010

Указатели

Указатель – это переменная особого рода: она содержит не само значение переменной, а *адрес* той области памяти, где искомое значение хранится.

Указатель – это одна из наиболее мощных возможностей языка Си и, как любое мощное средство, указатель может быть как чрезвычайно удобным инструментом, так и источником больших проблем, если используется неумело.

Для объявления указателя достаточно добавить к типу переменной символ «*»:

```
unsigned int * iptr;
char * arr[];
long * lptr;
```

Объявленные таким образом переменные **iptr**, **arr[]** и **lptr** будут указателями: **iptr** – указатель на число типа **int**, **arr** – *массив указателей* на символ, **lptr** – указатель на число типа **long**.

Чтобы обратиться к тем данным, на которые указывает переменная-указатель, следует предварить ее имя символом «*» (такая операция получила название «*разыменование указателя*»):

iptr – адрес области памяти, хранящей число типа **int**

***iptr** – значение этого числа

Очевидно, что использование указателей позволит получить доступ к любой произвольной области памяти. Именно в этом и кроется источник возможных проблем: стоит ошибиться в значении переменной-указателя, как последствия обращения к данным, на которые она указывает, становятся непредсказуемыми.

Выражения

Выражение – это запись алгоритма вычислений над *операндами* при помощи знаков математических, логических и прочих действий. Другими словами: выражение – есть формула вычислений, записанная средствами языка программирования.

В определении выражения использован новый термин: *операнд*.

Операнд – это выражение, над которым выполняется какое-либо действие. Как видите, операнд и выражение оказываются определенными друг через друга. Надеюсь, для образованного человека не составит труда представить себе, что же такое выражение на самом деле – для этого достаточно вспомнить, что такое математическое выражение.

В языке Си определены два типа операндов (выражений): математические и логические. Математическое выражение состоит из операндов, эквивалентных числам, и знаков математических операций. Логическое выражение – это выражение отношений, т.е. ответов на вопросы «**больше**?», «**меньше**?», «**равно**?» и т.д. Аналогично математическим, существуют логические действия и соответствующие им знаки логических операций.

Действия могут выполняться над двумя операндами или над одним операндом – *унарные* операции.

В Си определены следующие унарные математические операции:

Знак	Операция
–	Унарный минус, признак изменения знака числа на противоположный
+	Унарный плюс, означает сохранение знака числа без изменения (принципиального смысла не имеет)
++	Знак инкремента, увеличение операнда на 1
--	Знак декремента, уменьшение операнда на 1
~	Побитовая инверсия числа

Математических операций над двумя операндами больше:

Знак	Операция
+	Сумма операндов
–	Разность операндов
*	Произведение операндов
/	Частное операндов
%	Остаток от целочисленного деления операндов
|	Побитовая операция ИЛИ над операндами
&	Побитовая операция И над операндами
^	Побитовая операция ИСКЛЮЧАЮЩЕЕ ИЛИ над операндами
>>	Побитовый сдвиг влево
<<	Побитовый сдвиг вправо

Об операциях побитового сдвига следует упомянуть, что эти операции *сохраняют знак* операнда, если операнд имеет знаковый тип.

Унарная логическая операция всего одна:

Знак	Операция
!	Логическое отрицание (НЕ)

Логических операций над парой операндов больше:

Знак	Операция
||	Логическое ИЛИ
&&	Логическое И
>	Больше
<	Меньше
==	Эквивалентно
!=	Не эквивалентно
<=	Меньше или равно
>=	Больше или равно

Важно понимать, что для логических выражений определено лишь два значения результата: ИСТИНА и ЛОЖЬ, причем истинным в Си считается любое *не равное нулю* значение, а ложным – значение ноль.

Вот несколько примеров логических и арифметических выражений:

```
5 + 2 * 3 – арифметическое выражение, эквивалентное 11
(5 + 2) * 3 – арифметическое выражение, эквивалентное 21
5 + var * (3 + b) – арифметическое выражение с использованием переменных
(5 > 2) – логическое выражение со значением ЛОЖЬ
(2-7) >= -11 – логическое выражение со значением ИСТИНА
(2-7) >= 0 – логическое выражение со значением ЛОЖЬ
```

Знаки арифметических и логических операций называются по-другому *операторами*, однако понятие оператор более широкое, и включает в себя не только какое либо одно действие, но и целую группу действий, выполняемых по определенным правилам.

Особняком стоит операция, обозначаемая символом «**?**» – *условная операция*. Эта операция имеет достаточно сложную форму записи:

<**выражение1**> **?** <**значение1**> **:** <**значение2**>

Операция позволяет выбрать в качестве своего результата одно из значений – **значение1** или **значение2** в зависимости от того, истинно или ложно **выражение1**. В качестве обоих значений могут использоваться так же выражения.

Пример:

x > 5 ? 4 : 8 – если **x** будет меньше или равно 5, то результат операции будет **8**, а если x будет больше 5 – результат будет **4**.

Смысл в выражениях был бы невелик, если бы результат их вычислений было бы невозможно присваивать значениям переменных. Для присвоения значения переменной используется оператор ***присваивания***, обозначаемый символом «**=**»:

```
var = 5 + 2; // присвоить переменной var значение выражения 5 + 2, т.е. 7
```

Возвращаясь к описанию переменных, следует заметить, что можно присваивать значение переменной сразу в момент ее описания:

```
int var = 5; // описанная переменная var имеет тип int и значение 5
```

В операторе присваивания левее знака «=» должно находиться так называемое *леводопустимое* выражение, т.е. либо переменная, либо такое выражение, результатом которого будет значение указателя, который покажет место в памяти для хранения результата. Правее знака равенства может находиться любое *праводопустимое*

выражение, т.е. в сущности, любое выражение. Интересный момент возникает, если и слева и справа от знака равенства используется одна и та же переменная:

```
S = S + 5;
```

Такое выражение следует понимать так: «присвоить переменной S значение, равное сумме ее текущего значения и числа 5». Можно записать то же самое более коротким способом:

```
S += 5;
```

В данном примере использован двойной оператор «**+=**», называемый *присваивание с суммированием*. Значение этого оператора то же самое: увеличение значения переменной на число правее оператора. Кроме присваивания с суммированием допустимы аналогичные комбинации для присваивания с разностью, с умножением, делением или остатком от деления, записываемым соответственно так: «**–=**», «***=**», «**/=**» или «**%=**». Любой такой оператор «разворачивается» аналогично рассмотренному:

```
a *= 2          равносильно          a = a * 2
a /= 2          равносильно          a = a / 2
a -= 2          равносильно          a = a – 2
a %= 2          равносильно          a = a % 2
```

Несколько слов о присваивании значений переменным типа *указатель*. Как было сказано, указатель есть ни что иное, как адрес объекта в памяти, соответственно в качестве значения можно ему присваивать только адрес. Делается это при помощи унарного оператора взятия адреса **&** (не путать с побитовой операцией И !!!):

```
int *ptr;
int var;

ptr = &var;
```

В этом примере объявлен указатель **ptr** на переменную типа **int** и переменная **var** этого типа; далее происходит присваивание указателю значения адреса переменной **var**.

Теперь операторы

```
*ptr = 12;
```

и

```
var = 12;
```

приведут к одинаковому результату.

Иногда требуется обратиться к какому-либо участку памяти, который явно не является конкретной переменной, т.е. адрес участка известен, но соответствующего ему описанного в программе объекта нет. В этом случае следует использовать прием, названный *явным приведением типа*:

```
ptr = (int *)0x1000;
```

Пример показывает, как указателю **ptr** присваивается значение адреса **0x1000**, т.е. происходит «обман» бдительного компилятора: ему подсовывают указание воспринимать число, как адрес переменной типа **int**, хотя не известно, что именно находится по этому адресу фактически. Приведение типов приходится делать всегда, когда типы операндов не соответствуют желаемым.

Приоритеты операций

Каждая операция имеет свой приоритет, который определяет порядок вычисления операции в выражении: если слева направо следуют подряд несколько операций одинакового приоритета, порядок их вычисления не определен, но результат вычисления гарантированно будет верным, в противном случае вычисляются прежде операции с наибольшим приоритетом, а затем – с меньшим. Порядок вычислений может быть изменен при помощи круглых скобок: выражение в скобках вычисляется всегда прежде остальных, т.е. имеет наивысший приоритет.

В таблице перечислены все операции с указанием их приоритетов:

Операция	Приоритет
! ~ «унарные плюс и минус» ++ -- ? операция разыменования указателя *	13
* / %	12
+ - (	11
>> и <<	10
< <= > >=	9
== !=	8
&	7
^	6
|	5
&&	4
||	3
?	2
= *= /= += -= &= ^= |= <<= >>=	1

Так как запомнить все приоритеты довольно непросто, желательно на первых порах (и не возбраняется вообще всегда) указывать порядок вычислений при помощи скобок.

Использование операторов **++** и **--** совместно с указателями приводит к необходимости помнить о порядке выполнения операций:

```
// вариант 1
ptr = &temp;
var = *ptr++;
// вариант 2
ptr = &temp;
var = (*ptr)++;
```

Оба варианта дадут одно и то же значение **var**, но совсем разные значений для **temp** и **ptr**: в первом случае переменная **temp** не изменится, но увеличится на 1 значение указателя **ptr**; во втором случае неизменным останется значение указателя, но изменится содержимое переменной, на которую он указывает, т.е. **temp** увеличится на 1.

Операторы **++** и **--** с указателями работают особым образом: они изменяют значение указателя на величину, равную размеру типа элемента, на который указатель указывает. То есть если указатель объявлен как **int *ptr**, то **ptr++** изменит значение указателя на 2, если **long *ptr**, то **ptr++** увеличит указатель на 4 и т.д. Увеличение указателя на 1 происходит только для типа указателя (**void ***) («абстрактный указатель», просто адрес байта) или (**char ***).

В любом выражении допустимо комбинировать логические и арифметические выражения, в этом случае следует считать, что логическому значению ***ЛОЖЬ*** соответствует арифметическое значение ноль, а значению ***ИСТИНА*** – арифметическое значение 1:

```
var = (5 < 2) + 2;      // переменной var будет присвоено значение 2
var = (5 >= 2) + 2;     // переменной var будет присвоено значение 3
```

Особое внимание следует обращать на приоритет операций:

```
var = (5 > 2) + 2;      // var = 3
var = 5 > 2 + 2;        // var = 0
```

Во втором выражении более высокий приоритет имеет арифметическая операция сложения, поэтому переменной **var** будет присвоено значение выражения **5 > 4**, что ЛОЖНО, т.е. **var** будет равна **0**.

Наиболее простым и универсальным советом будет использование скобок везде, где необходимо точно гарантировать порядок вычислений – в этом случае можно считать (конечно, лишь условно!) все операторы равноприоритетными, а порядок вычислений определять лишь с помощью скобок. Возможно, это приведет к некоторой излишней «скобочности» программы, зато гарантирует, что планы программиста не разойдутся с мнением компилятора.

Упомянутые ранее унарные операторы инкремента и декремента – одна из интересных особенностей языка Си. Эти операторы могут применяться только к операнду-переменной, и ни в коем случае не к операнду-константе. Но главное, они могут быть ***префиксными*** и ***постфиксными***, т.е. записываться либо *слева* от переменной, либо *справа*:

```
int var = 1;
var++;          // теперь var = 2
--var;          // теперь var = 1
var--;          // var = 0
```

Казалось бы, к чему два способа выполнить одно и то же? Все дело в том, как будет использован результат оператора при вычислении выражений. Вот пример:

```
int var = 5;
int b = var++ - 5;      // var становится равным 6
int c = ++var - 5;      // var становится равным 7
```

Переменная **b** окажется равной 0, а переменная **c** будет равна 2. Все дело в том, что при ***постфиксном*** операторе **++** сначала используется значение переменной в выражении, а потом выполняется оператор инкремента, а при ***префиксной*** записи – сначала выполняется оператор инкремента, а потом уже обновленное значение переменной используется при вычислении выражения.

Операторы побитового сдвига << и >> выполняют перемещение битов операнда слева на количество разрядов, заданное выражением справа. Как уже было сказано, эти операторы сохраняют знак левого операнда, если операнд – число со знаком. Заполнение «освобождаемых» разрядов для беззнаковых чисел происходит нулями.

Унарные логические операции, и операция побитовой инверсии имеют лишь один вариант вычисления. Но, к сожалению, на этом простые операторы заканчиваются, все прочие операторы имеют более сложную форму записи и, соответственно, выполняют более сложные действия, заслуживая отдельного рассмотрения.

Тип результата выражения

Выражения могут состоять из операндов различных типов, например, не возбраняется осуществлять суммирование переменных типа **char** и **long**, **double** и **unsigned int** и т.д. И результат этих вычислений так же может быть присвоен переменной любого типа. Как же ведет себя компилятор в этом случае?

Рассмотрим следующий пример:

```
int x1 = 20000, x2 = 15;
long res;

res = x1 * x2;
```

Переменной **res** (длинное целое) присваивается результат произведения двух чисел (**int**), значения переменных укладываются в допустимые диапазоны, ожидается, что значение **res** будет **300000**. Однако, это ошибочное предположение.

Стандартом Си определено, что тип *результата выражения* определяется по набольшему типу входящего в него операнда, т.е. в данном случае результат **x1 * x2** будет иметь тип **int**. Разумеется, произведение чисел вызовет переполнение результата, и все, что «не влезло» в размер, будет отброшено, т.е. в результате произведение будет равно **-27680**.

Такой странный результат получился из-за несоответствия типа результата выражения и типа операндов. Чтобы получить верный результат, нужно использовать один из операндов типа **long**:

```
res = (long)x1 * x2;
// или так
res = x1 * (long)x2;
```

Такая запись показывает, что компилятор должен воспринимать операнд **x1**, как число типа **long**. Принудительное указание в круглых скобках нового типа для выражения называется ***явным преобразованием*** или ***приведением*** типа. Ошибкой будет попытка задать тип для результата всего выражения:

```
res = (long) (x1 * x2);
```

Такая запись ничем не отличается от первоначальной и даст неверный результат (т.к. в этой записи явно указано то преобразование, которое компилятор и так делает автоматически).

Компилятор всегда приводит автоматически типы всех операндов в выражении к наибольшему типу, т.е. от меньшего типа к большему. Важно помнить, что если в выражении все операнды типа char, то они будут приведены к типу int, если нет явного преобразования типов. При помощи приведения типа программист может изменить его поведение, что чревато большими ошибками.

Операторы

Оператор if

Условный оператор if позволяет выполнить группу операторов лишь в случае, если какое-то логическое выражение имеет значение ИСТИНА. Дополнительно может быть определена группа операторов, выполняемых в случае, если заданное выражение имеет значение ЛОЖЬ.

Оператор if может быть описан по одному из следующих шаблонов:

1. **if** (<**проверка**>) <**оператор1**>;
2. **if** (<**проверка**>) <**оператор1**>; **else** <**оператор2**>;
3. **if** (<**проверка**>) { <**список операторов1** ;>}
4. **if** (<**проверка**>) { <**список операторов1** ;>} **else** {**список операторов2** ;}

Здесь **проверка** – это логическое выражение, проверяемое на истинность, **оператор1** – оператор, выполняемый только в случае истинности проверки, **оператор2** выполняется только в случае ложности проверки. Под списком операторов подразумевается любая последовательность любых операторов, причем, как и одиночные, эти списки выполняются лишь в соответствующем случае в зависимости от результата проверки.

Обратите внимание, что для 3 и 4 форм оператора **if** наличие завершающей оператор точки с запятой не является необходимым. В данном случае закрывающая фигурная скобка однозначно свидетельствует о завершении оператора, в то время как в списках каждый из операторов должен завершаться точкой с запятой (если, конечно, оператор в списке не ограничивается фигурными скобками).

Шаблоны оператора **if** не накладывают никаких ограничений на то, какие операторы могут быть на месте **оператра1** или **оператора2** или же в соответствующих списках. Это означает, что среди них так же может использоваться оператор **if**. В этом случае может возникнуть проблема с определением принадлежности части оператора **else**:

```
if (a < 2) if (c > 5) result = 5; else result = 0;
```

В этом примере стоит задуматься, к какому из операторов **if** относится часть **else**? Язык Си однозначно вводит следующее правило: всякий **else** относится к ближайшему предыдущему **if**, не имеющему своего **else**. Таким образом, в вышеприведенном примере **else** относится к оператору **if(c > 5)**.

Очевидно, что запись оператора в одну строку не способствует улучшению восприятия программы. Благодаря тому, что язык Си допускает использование любых «пробельных» разделителей, гораздо более удачной следует считать следующую форму записи:

```
if (a < 2)
        if (c > 5)
                result = 5;
        else
                result = 0;
```

Благодаря использованию табуляций вышеприведенный пример даже визуально показывает принадлежность **else** соответствующему оператору **if**.

Следует отметить, что пример показывает одну из проблем, характерную для оператора **if** – так называемую *неполную проверку* вариантов. Она заключается в следующем: если, например, переменная **a** равна 5, то переменная **result** оказывается с неопределенным значением. Возможно, она получит (или получила) нужное значение в другом операторе, возможно переменная **a** никогда не получит значение больше 1 (но тогда, к чему вообще проверка?!), однако если это не так – проблем не миновать.

Итак, при использовании оператора **if** желательно не допускать двусмысленностей его поведения при различных результатах проверяемых выражений. Так же не лишним будет использование фигурных скобок для определения границ списка операторов, даже если фактически используется единственный оператор.

Оператор switch

Анализ разных значений переменной при помощи оператора if может вылиться в «многоэтажную» конструкцию типа такой:

```
if (x == 0) x = 5;
else
        if (x == 1) x = 12;
        else
                if (x == 3) x = 36;
                // и т.д. до перебора всех вариантов значения x
```

Это очень неудобная конструкция. Заменить ее призван оператор **switch**, который записывается по следующему шаблону:

switch (<выражение>){
case <значение1>: [операторы1];
case <значение2> : [операторы2];
// и т.д. до конца вариантов значений
[default : [операторы по умолчанию]];
}

Здесь **выражение** – это выражение, значения которого анализируются, **значение1** – это один из вариантов анализируемого результата выражения, **операторы1** - соответствующая этому варианту обработка, **значение2** и **операторы2** – это соответственно следующее анализируемое значение и соответствующая ему обработка и т.д. **Операторы по умолчанию** выполняются в том случае, если выражение имеет значение, не соответствующее ни одному из определенных в строке с ключевым словом **case**. Операторы по умолчанию и ключевое слово **default** могут отсутствовать.

Результат анализируемого выражения обязательно должен быть одного из целочисленных типов, а в качестве значений case обязательно должны использоваться константы или выражения, вычисляемые на основе констант. Допускается использовать символы (в апострофах).

Оператор действует следующим образом: вычисляется значение выражения, результат проверяется на совпадение с одним из указанных в строчках case. Если найдено совпадение, то, начиная с этого места и вплоть до конца оператора (т.е. до закрывающей фигурной скобки), выполняются все обозначенные операторы, в том числе и те, которые относятся к другим, «нижеследующим» за найденным значениям. Если совпадений не найдено, то выполнятся операторы по умолчанию, если они определены, в противном случае не будет выполнено никаких действий. Пример:

```
int res;
switch (x){
        case 0 : res = 1;
        case 1 : res = 2;
        default : res = 3;
}
```

Если x в этом примере равен 5, то значение переменной res станет равно 3. Однако, если x будет равен 0, то res все равно окажется равным 3, т.к. поочередно выполнятся все операторы, начиная с того, что указан в строке case 0. Чтобы этого бессмысленного поведения не происходило, введен особый оператор *break*, который вызывает прекращение работы оператора switch *досрочно*, т.е. этот оператор является последним исполняемым оператором внутри switch. Таким образом, правильная запись желаемого алгоритма должна быть такой:

```
int res;
switch (x){
        case 0 : res = 1; break;
        case 1 : res = 2; break;
        default : res = 3;
}
```

В этом случае для x равного 0, 1 и 5 будут получены соответственно значения res 1, 2 и 3. Для последнего варианта наличие «прекращающего» оператора не требуется, это очевидно.

Оператор for

При реализации многих алгоритмов бывает необходимо повторить некую последовательность действий несколько раз. Такое повторение получило наименование цикла, а оператор, для соответствующего действия – оператором цикла.

В Си имеется несколько операторов цикла, один из которых – оператор **for**. Шаблон описания оператора следующий:

for ([**список операторов1**]; [**условие продолжения**]; [**список операторов 2**]) [**тело цикла**];

Здесь **список операторов1** – перечень операторов, разделенных запятыми, которые должны быть выполнены *перед* началом цикла; **условие продолжения** – логическое выражение, *истинное* значение которого есть непременное условие очередной итерации цикла; **список операторов2** – перечень операторов, разделенных запятыми, выполняемых *после* каждой итерации цикла; **тело цикла** – это один оператор или ограниченная фигурными скобками последовательность операторов, выполняемых на каждой итерации.

Самое важное, что любая из этих частей оператора **for** может отсутствовать, о чем свидетельствуют квадратные скобки в шаблоне.

Пример некоторых вариантов оператора цикла for:

```
for (i = 0, s = 0; i < 10; s += i++);
for (;;);
for (;i < 10;);
for (i = 0, j = 9, s = 0; i < 10; i++, j–) s += arr[i][j];
```

Первый цикл примера вычисляет сумму всех чисел от 0 до 9 и помещает результат в переменной s. Рассмотрим более подробно, как это происходит.

Сначала выполняются операторы i=0 и s = 0, которые инициализируют (ранее объявленные) переменные i и s. Если в момент объявления эти переменные уже проинициализированы, эта часть оператора for может отсутствовать. Затем происходит проверка условия выполнения итерации цикла, т.е. проверка i < 10. Разумеется, это истинное выражение, и, значит, происходит выполнение тела цикла – очередная (первая) итерация. В рассматриваемом примере тело цикла отсутствует, поэтому просто выполняется завершающая часть, которая состоит из оператора s += i++, который одновременно выполняет 2 действия: накапливает в переменной s сумму значений переменной i, а затем увеличивает значение i на 1. После этих действий снова осуществляется проверка условия, затем очередная итерация и т.д. до тех пор, пока значение i не станет равным 10. В этот момент условие i < 10 станет ложным, и выполнение оператора for закончится.

Продолжим рассмотрение примеров операторов for. Второй цикл, единожды начавшись, никогда не закончится, т.е. это бесконечный оператор, или бесконечный цикл. В этом случае отсутствие выражения проверки условия трактуется как истинное выражение, т.е. «нет требований к условию – значит, любые варианты подходят».

Третий пример показывает оператор цикла, который может быть бесконечным, если переменная i < 10. Т.е. для того, чтобы этот цикл не смог «завесить» программу, необходимо, чтобы либо к началу цикла переменная i уже содержала значение 10 или более, либо каким-то образом во время выполнения цикла значение i должно быть изменено.

Наконец, четвертый пример показывает, как вычислить сумму «правой» диагонали матрицы arr размером 10x10.

Следует отметить, что допускается[11] внутри списка **операторов1** использовать определение переменных, однако, определенные таким образом переменные, считаются существующими только пока цикл выполняется, и «исчезают» после его завершения:

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

Оператор **for** имеет довольно много вариантов и возможностей, однако не стоит увлекаться предоставляемой гибкостью. Чем более простая форма оператора будет избрана, тем меньше вероятность ошибки программирования:

```
// первый способ
for (int i, j=9, s; i < 10; s += arr[i++][j–]);

// второй способ
int i=0, s=0, o = 9;
for (; i<10; i++) {
        s += arr[i][j];
        j–;
}
```

[11] Такое допущение для компилятора WinAVR возможно лишь в том случае, если включен режим соответствия стандарту C99 с расширениями GNU.

В этом примере приведены два варианта ранее рассмотренного подсчета суммы «правой» диагонали матрицы, однако первый способ требует гораздо больше умственных усилий для понимания написанного, чем второй.

Оператор while

Оператор **for** удобен, если требуется повторить какие-либо действия определенное число раз. Но иногда число повторений неизвестно, хотя известно условие завершения цикла (например, при решении уравнения методом последовательных приближений). В этом случае можно использовать оператор цикла по условию **while**.

Шаблон этого оператора следующий:

while (**условие**) [**тело цикла**];

Здесь **условие** – логическое выражение, истинное значение которого является условием продолжения цикла, **тело цикла** – это либо один оператор, либо ограниченная фигурными скобками последовательность операторов.

Важная особенность **while** в том, что если к моменту его начала условие ложно – тело цикла не выполнится ни разу.

Пример операторов **while**:

```
while (1);
while (PORT == 2) sum++;
```

Первый пример демонстрирует бесконечный цикл, т.к. ненулевое значение равносильно истинному логическому выражению. Второй пример опрашивает значение переменной **PORT** и ведет подсчет в переменной **sum** числа таких опросов до тех пор, пока значение переменной равно двум[12].

По сравнению с оператором **for**, цикл **while** допускает значительно меньше вариантов «оформления». Очевидно, что цикл **for** вполне в состоянии заменить цикл **while**:

```
// эквивалент бесконечного цикла
for (;;);
// эквивалент опроса и подсчета числа этих опросов
for (; PORT == 2; sum++);
```

Тем не менее, рекомендуется использовать **while** там, где это оправдано логикой алгоритма, это позволит улучшить «читабельность» программы.

Оператор do

Как было показано, тело оператора **while** может быть не выполнено ни разу, если условие его продолжения сразу ложно. Но иногда требуется, чтобы при прочих равных условиях тело цикла всегда выполнялось хотя бы один раз. Для этих целей применяется оператор **do**, записываемый по следующему шаблону:

do [**тело цикла**] **while** (**условие**);

Здесь, как и ранее, **тело цикла** – это либо один оператор, либо заключенная в фигурные скобки последовательность операторов, а **условие** – это логическое выражение, определяющее условие продолжения итераций цикла. Как и ранее, тело цикла может отсутствовать.

В остальном цикл **do** аналогичен циклу **while**.

Оператор break

Оператор **break** служит для досрочного завершения операторов цикла (любых) и оператора **switch**. Исполнение этого оператора равносильно переходу к оператору, следующему за «телом» упомянутых операторов.

Пример:

```
int arr[10];
int found=-1;
for (int i = 0; i < 10; i++) if (arr[i] == 12) { found = i; break;}
```

В этом примере осуществляется поиск в массиве **arr** элемента, равного числу 12. Как только такой элемент будет найден, цикл завершится досрочно, и значение **i** будет сохранено в переменной **found**. Если после завершения цикла значение **found** будет равно -1, это будет означать, что элементов 12 в массиве нет, в противном случае **found** будет равно индексу первого найденного элемента.

[12] То, каким образом значение переменной PORT может измениться – не рассматривается в данном контексте, предполагается пока, что все-таки такое изменение возможно.

Оператор continue

Оператор **continue** служит для досрочного начала очередной итерации цикла, т.е. он обеспечивает немедленный переход к проверке условия для циклов **do** или **while**, или к переходу к выполнению *последней группы операторов*, указанных в круглых скобках, для цикла **for**.

Этот оператор служит для пропуска части операторов цикла, если на определенной итерации их выполнять не следует. Пример:

```
int arr[10];
int found, sum;
for (int i = 0; i < 10; i++){
        if (arr[i] == 12){
                found++;
                continue;
        }
        sum += arr[i];
}
```

В этом примере осуществляется подсчет суммы **sum** всех значений массива **arr**, не равных 12, и одновременно подсчет в **found** количества элементов, равных 12.

Оператор goto

Последний оператор языка Си – оператор безусловного перехода goto. Шаблон оператора такой:

goto <**метка**>;

Метка – это идентификатор, при объявлении завершаемый двоеточием. Этот идентификатор служит для обозначения определенного места в программе, где метка объявлена (отождествляется с адресом исполняемого кода в памяти программ). При использовании в операторе **goto** метка указывается уже без двоеточия.

Оператор **goto** вызывает безусловное продолжение исполнения программы с указанной метки.

Пример использования меток и оператора **goto**:

```
        if (a < 5) goto m1;
        a = 0;
        goto m2;
m1:
        a = 25;
m2:
```

Этот пример показывает реализацию следующего алгоритма: если значение переменной **a** меньше пяти, то присвоить переменной значение **25**, в противном случае обнулить переменную **a**.

Использование оператора **goto** в программах считается дурным тоном среди программистов. Как правило, программы с этими операторами более запутаны, труднее отлаживаются, таят больше потенциальных возможностей для ошибок. Метка может находиться почти в любом[13] месте программы, и соответственно, оператор **goto** может заставить программу изменить нормальный ход непредсказуемым образом, если программист случайно забудет вовремя убрать или изменить нужную метку. Доказано, что любой алгоритм может быть реализован без использования **goto**, лишь с помощью других операторов языка Си. Например, только что рассмотренный алгоритм элементарно и гораздо красивее реализуется так:

```
if (a < 5)
        a = 25;
else
        a = 0;
```

Продолжение в №3/2010

[13] Об ограничениях на размещение меток см. главу «Структура программы».