

3. Инструкции управления. Основные циклы

Порядок, в котором выполняются вычисления, определяется инструкциями управления. Мы уже встречались с наиболее распространенными управляющими конструкциями такого рода в предыдущих примерах; здесь мы завершим их список и более точно определим рассмотренные ранее.

3.1. Инструкции и блоки

Выражение, скажем `x = 0`, или `i++`, или `printf (...)`, становится *инструкцией*, если в конце его поставить точку с запятой, например:

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

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

Фигурные скобки `{` и `}` используются для объединения объявлений и инструкций в составную инструкцию, или блок, чтобы с точки зрения синтаксиса эта новая конструкция воспринималась как одна инструкция. Фигурные скобки, обрамляющие группу инструкций, образующих тело функции, — это один пример; второй пример — это скобки, объединяющие инструкции, помещенные после `if`, `else`, `while` или `for`. (Переменные могут быть объявлены внутри *любого* блока, об этом разговор пойдет в главе 4.) После правой закрывающей фигурной скобки в конце блока точка с запятой не ставится.

3.2. Конструкция if-else

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

```
if (выражение)  
    инструкция1
```

```
else
```

причем `else`-часть может и отсутствовать. Сначала вычисляется выражение, и, если оно истинно (т. е. отлично от нуля), выполняется *инструкция1*. Если выражение ложно (т. е. его значение равно нулю) и существует `else`-часть, то выполняется *инструкция2*.

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.

Отсутствие `else`-части в одной из вложенных друг в друга `if`-конструкций может привести к неоднозначному толкованию записи. Эту неоднозначность разрешают тем, что `else` связывают с ближайшим `if`, у которого нет своего `else`. Например, в

```
if (n > 0)  
    if (a > b)  
        z = a;  
    else  
        z = b;
```

`else` относится к внутреннему `if`, что мы и показали с помощью отступов. Если нам требуется иная интерпретация, необходимо должным образом расставить фигурные скобки:

```
if (n > 0) {
    if (a > b)
        z = a;
}
else
    z = b;
```

Ниже приводится пример ситуации, когда неоднозначность особенно опасна:

```
if (n >= 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf ( " . . . " )
            ; return i;
        }
else /* НЕВЕРНО */
    printf("ошибка - отрицательное n\n");
```

С помощью отступов мы недвусмысленно показали, что нам нужно, однако компилятор не воспримет эту информацию и отнесет `else` к внутреннему `if`. Искать такого рода ошибки особенно тяжело. Здесь уместен следующий совет: вложенные `if` обрамляйте фигурными скобками.

Кстати, обратите внимание на точку с запятой после `z = a` в

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

Здесь она обязательна, поскольку по правилам грамматики за `if` должна следовать инструкция, а выражение-инструкция вроде `z = a;` всегда заканчивается точкой с запятой.

3.3. Конструкция `else-if`

Конструкция

```
if (выражение)
    инструкция
else if (выражение)
    инструкция
else if (выражение)
    инструкция
else if (выражение)
    инструкция
else
    инструкция
```

встречается так часто, что о ней стоит поговорить особо. Приведенная последовательность инструкций `if` — самый общий способ описания многоступенчатого принятия решения. *Выражения* вычисляются по порядку; как только встречается *выражение* со значением "истина", выполняется соответствующая ему *инструкция*; на этом последовательность проверок завершается. Здесь под словом *инструкция* имеется в виду либо одна инструкция, либо группа инструкций в фигурных скобках.

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.

В качестве иллюстрации трехпутевого ветвления рассмотрим функцию бинарного поиска значения `x` в массиве `v`. Предполагается, что элементы `v` упорядочены по возрастанию. Функция выдает положение `x` в `v` (число в пределах от 0 до `n-1`), если `x` там встречается, и `-1`, если его нет.

При бинарном поиске значение `x` сначала сравнивается с элементом, занимающим серединное положение в массиве `v`. Если `x` меньше, чем это значение, то областью поиска становится "верхняя" половина массива `v`, в противном случае — "нижняя". В любом случае следующий шаг — это сравнение с серединным элементом отобранной половины. Процесс "уполовинивания" диапазона продолжается до тех пор, пока либо не будет найдено значение, либо не станет пустым диапазон поиска. Запишем функцию бинарного поиска:

```
/* binsearch: найти x в v[0] <= v[1] <= ... <= v[n-1]  
*/ int binsearch(int x, int v[], int n)  
{  
    int low, high, mid;  
  
    low = 0;  
    high = n - 1 ;  
    while (low <= high) {  
        mid = (low + high) / 2;  
        if (x < v[mid])  
            high = mid - 1; else  
            if (x > v[mid]) low  
                = mid + 1 ;  
        else /* совпадение найдено  
            */ return mid;  
    }  
    return -1; /* совпадения нет */  
}
```

Основное действие, выполняемое на каждом шаге поиска, — сравнение значения `x` (меньше, больше или равно) с элементом `v[mid]`; это сравнение естественно поручить конструкции `else-if`.

Упражнение 3.1. В нашей программе бинарного поиска внутри цикла осуществляются две проверки, хотя могла быть только одна (при увеличении числа проверок вне цикла). Напишите программу, предусмотрев в ней одну проверку внутри цикла. Оцените разницу во времени выполнения.

3.4. 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.

В главе 1 мы написали программу, подсчитывающую число вхождений в текст каждой цифры, символов-разделителей (пробелов, табуляций и новых строк) и всех остальных символов. В ней мы использовали последовательность `if...else if...else`. Теперь приведем вариант этой программы с переключателем `switch`:

```
#include <stdio.h>

main() /* подсчет цифр, символов-разделителей и прочих символов */
{
    int c, i, nwhite, nother, ndigit[10];

    nwhite = nother = 0; for
    (i = 0; i < 10; i++)
        ndigit[i] = 0;
    while ((c = getchar()) != EOF)
    { switch (c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            ndigit[c - '0']++;
            break;
        case ' ':
        case '\n':
        case '\t':
            nwhite++;
            break;
        default:
            nother++;
            break;
    }
    }
    printf ("цифр =");
    for (i = 0; i < 10; i++) printf
        (" %d", ndigit[i]);
    printf ("", символов-разделителей = %d, прочих = %d\n", nwhite, nother);
    return 0;
}
```

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. Инструкция `break` используется также для принудительного выхода из циклов `while`, `for` и `do-while` (мы еще поговорим об этом чуть позже).

"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 при

изменении 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.

Добрый вам совет: 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`.

Упражнение 3.2. Напишите функцию `escape(s, t)`, которая при копировании текста из `t` в `s` преобразует такие символы, как новая строка и табуляция в "видимые последовательности символов" (вроде `\n` и `\t`). Используйте инструкцию `switch`. Напишите функцию, выполняющую обратное преобразование эскейп-последовательностей в настоящие символы.

3.5. Циклы `while` и `for`

Мы уже встречались с циклами `while` и `for`. В цикле

```
while (выражение)
    инструкция
```

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

Инструкция `for`

```
for (выр1; выр2;
     выр3) инструкция
```

эквивалентна конструкции

```
выр1;
while (выр2) {
    инструкци
    я выр3;
}
```

если не считать отличий в поведении инструкции `continue`, речь о которой пойдет в параграфе 3. 7.

С точки зрения грамматики три компоненты цикла `for` представляют собой произвольные выражения, но чаще *выр1* и *выр3* — это присваивания или вызовы функций, а *выр2* — выражение отношения. Любое из этих трех выражений может отсутствовать, но точку с запятой опускать нельзя. При отсутствии *выр1* или *выр3* считается, что их просто нет в конструкции цикла; при отсутствии *выр2* 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`.

Какой цикл выбрать: `while` или `for` — это дело вкуса. Так, в

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ; /* обойти символы-разделители */
```

нет ни инициализации, ни пересчета параметра, поэтому здесь больше подходит `while`.

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++)
    ...
```

Это похоже на `DO`-циклы в Фортране и `for`-циклы в Паскале. Сходство, однако, не вполне точное, так как в Си индекс и его предельное значение могут изменяться внутри цикла, и значение индекса `i` после выхода из цикла всегда определено. Поскольку три компоненты цикла могут быть произвольными выражениями, организация `for`-циклов не ограничивается только случаем арифметической прогрессии. Однако включать в заголовок цикла вычисления, не имеющие отношения к инициализации и инкрементированию, считается плохим стилем. Заголовок лучше оставить только для операций управления циклом.

В качестве более внушительного примера приведем другую версию программы `atoi`, выполняющей преобразование строки в ее числовой эквивалент. Это более общая версия по сравнению с рассмотренной в главе 2, в том смысле, что она игнорирует левые символы-разделители (если они есть) и должным образом реагирует на знаки `+` и `-`, которые могут стоять перед цифрами. (В главе 4 будет рассмотрен вариант `atof`, который осуществляет подобное преобразование для чисел с плавающей точкой.)

Структура программы отражает вид вводимой информации:

*игнорировать символы-разделители, если они
есть получить знак, если он есть взять целую
часть и преобразовать ее*

На каждом шаге выполняется определенная часть работы и четко фиксируется ее результат, который затем используется на следующем шаге. Обработка данных заканчивается на первом же символе, который не может быть частью числа.

```
#include <ctype.h>

/* atoi: преобразование s в целое число; версия 2 */
int atoi(char s[])
{
    int i, n, sign;

    /* игнорировать символы-разделители */
    for (i = 0; isspace(s[i]); i++)
        ;
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-') /* пропуск знака */
        i++;
    for (n = 0; isdigit(s[i]); i++)
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

Заметим, что в стандартной библиотеке имеется более совершенная функция преобразования строки в длинное целое (`long int`) — функция `strtol` (см. параграф 5 приложения В).

Преимущества, которые дает централизация управления циклом, становятся еще более очевидными, когда несколько циклов вложены друг в друга. Проиллюстрируем их на примере сортировки массива целых чисел методом Шелла, предложенным им в 1959 г. Основная идея этого алгоритма в том, что на ранних стадиях сравниваются далеко отстоящие друг от друга, а не соседние элементы, как в обычных перестановочных сортировках. Это приводит к быстрому устранению массовой неупорядоченности, благодаря чему на более поздней стадии остается меньше работы. Интервал между сравниваемыми элементами постепенно уменьшается до единицы, и в этот момент сортировка сводится к обычным перестановкам соседних элементов. Программа `shellsort` имеет следующий вид:

```
/* shellsort: сортируются v[0] ... v[n-1] в возрастающем порядке
*/ void shellsort (int v[], int n)
{
    int gap, i, j, temp;
    for (gap = n/2; gap > 0; gap /=
        2) for (i = gap; i < n; i++)
        for (j = i - gap; j >= 0 && v[j] > v[j + gap]; j -= gap)
        { temp = v[j];
          v[j] = v[j + gap];
          v[j + gap] = temp;
        }
}
```

Здесь использованы три вложенных друг в друга цикла. Внешний управляет интервалом `gap` между сравниваемыми элементами, сокращая его путем деления пополам от `n/2` до нуля. Средний цикл перебирает элементы. Внутренний — сравнивает каждую пару элементов, отстоящих друг от друга на расстоянии `gap`, и переставляет элементы в неупорядоченных парах. Так как `gap` обязательно сведется к единице, все элементы в конечном счете будут упорядочены. Обратите внимание на то, что универсальность цикла `for` позволяет сделать внешний цикл по форме похожим на другие, хотя он и не является арифметической прогрессией.

Последний 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 несколько выражений, например вести два индекса параллельно. Продемонстрируем это на примере функции `reverse(s)`, которая "переворачивает" строку `s`, оставляя результат в той же строке `s`:

```
#include <string.h>

/* reverse: переворачивает строку s (результат в s)
*/ void reverse(char s[])
{
    int c, i, j;
    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
```

Запятые, разделяющие аргументы функции, переменные в объявлениях и пр. не являются операторами-запятыми и не обеспечивают вычислений слева направо.

Запятыми как операторами следует пользоваться умеренно. Более всего они уместны в конструкциях, которые тесно связаны друг с другом (как в `for`-цикле программы `reverse`), а также в макросах, в которых

многоступенчатые вычисления должны быть выражены одним выражением. Запятой-оператором в программе `reverse` можно было бы воспользоваться и при обмене символами в проверяемых парах элементов строки, мысля этот обмен как одну отдельную операцию:

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

3.6. 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, как, например, в функции `itoa` (обратной по отношению к `atoi`), преобразующей число в строку символов. Выполнить такое преобразование оказалось несколько более сложным делом, чем ожидалось, поскольку простые алгоритмы генерируют цифры в обратном порядке. Мы остановились на варианте, в котором сначала формируется обратная последовательность цифр, а затем она реверсируется.

```
/* itoa: преобразование n в строку s
*/ void itoa (int n, char s[])
{
    int i, sign;
    if ((sign = n) < 0) /* сохраняем знак */
        n = -n; /* делаем n положительным */
    i = 0;
    do { /* генерируем цифры в обратном порядке */
        s[i++] = n % 10 + '0'; /* следующая цифра */
    } while ((n /= 10) > 0); /* исключить ее
    */ if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

Конструкция `do-while` здесь необходима или по крайней мере удобна, поскольку в `s` посылается хотя бы один символ, даже если `n` равно нулю. В теле цикла одну инструкцию мы выделили фигурными скобками (хотя они и избыточны), чтобы неискушенный читатель не принял по ошибке слово `while` за начало цикла `while`.