

## Методические указания

### Тематическое занятие 4 **Работа с циклическими конструкциями.**

## Содержание

<b>Вложенный цикл</b> .....	1
<i>Вложенность</i> .....	1
<i>Задачи на перебор всех вариантов</i> .....	2
<i>Оптимизация программы</i> .....	3
<b>Операторы перехода</b> .....	3
<i>Оператор break</i> .....	3
<i>Оператор continue</i> .....	4
<i>Оператор goto</i> .....	5
<b>Переполнение и системно-зависимые константы</b> .....	5
<i>Границы целочисленных типов</i> .....	5
<i>Переполнение</i> .....	6
<i>Отслеживание переполнения</i> .....	6
<b>Упражнения</b> .....	7
<i>Упражнение 4.1</i> .....	7
<i>Упражнение 4.2</i> .....	7
<i>Упражнение 4.3</i> .....	7

## Вложенный цикл

### **Вложенность**

В теле любого оператора цикла (*внешнего*) могут находиться другие операторы цикла (*внутренние*). Все операторы внутреннего цикла должны полностью располагаться в теле внешнего цикла.

Пример вложенного цикла `for`:

```

for (ВыражИниц1; ВыражУсл1; ВыражИзмен1) {
    Оператор
    Оператор
    for (ВыражИниц2; ВыражУсл2; ВыражИзмен2) {
        Оператор
        Оператор
        Оператор
    }
    Оператор
    Оператор
}

```

Внутренний цикл выполняется полностью на каждой итерации внешнего цикла.

Пример. Вывод на экран таблицы умножения:

```
#include <stdio.h>
int main(void) {
    int i, j;    /* счетчики */
    for (i=1; i<=9; i++) {    /* внешний цикл */
        for (j=1; j<=9; j++)    /* внутренний цикл */
            printf("%d*%d=%2d  ", i, j, i*j);
        printf("\n");
    }
    return 0;
}
```

## Задачи на перебор всех вариантов

**Старинная задача о покупке скота.** Сколько можно купить быков, коров и телят (бык стоит 10 рублей, корова – 5 рублей, теленок – 0,5 рубля), если на 100 рублей надо купить 100 голов скота.

Обозначим:  $b$  – количество быков,  $k$  – количество коров,  $t$  – количество телят. Тогда условие задачи можно записать в виде системы из двух уравнений:  $10b+5k+0,5t=100$  и  $b+k+t=100$ . Преобразуем первое:  $20b+10k+t=200$ .

Запишем ограничения. На 100 рублей можно купить:

- не более 10 быков,  $0 \leq b \leq 10$ ;
- не более 20 коров,  $0 \leq k \leq 20$ ;
- не более 200 телят,  $0 \leq t \leq 200$ .

```
#include <stdio.h>
int main(void) {
    int b, k, t;
    for (b=0; b<=10; b++)
        for (k=0; k<=20; k++)
            for (t=1; t<=200; t++)
                if ((20*b+10*k+t==200)&&( b+k+t==100))
                    printf("быков %d, коров %d, телят %d\n ",b,k,t);
    return 0;
}
```

В данной программе значение переменной  $b$  изменяется 11 раз,  $k$  – 21 раз,  $t$  – 201 раз. Таким образом, условие в операторе `if` проверяется  $11 \times 21 \times 201 = 46431$  раз. Но если известно количество быков и коров, то количество телят можно вычислить по формуле  $t=100-(b+k)$ . Цикл по переменной  $t$  исключается:

```
#include <stdio.h>
int main(void) {
    int b, k, t;
    for (b=0; b<=10; b++)
        for (k=0; k<=20; k++) {
            t=100-(b+k);
            if (20*b+10*k+t==200)
                printf("быков %d, коров %d, телят %d\n",b,k,t);
        }
    return 0;
}
```

При этом решении условие проверяется  $11 \times 21 = 231$  раз, уменьшается процессорное время, необходимое на обработку программы. В данной задаче количество проверок можно еще уменьшить.

## Оптимизация программы

При вычислениях часто используются циклы, содержащие одинаковые операции для каждого слагаемого. Например, общий множитель:

```
sum=0;
for (i=1; i<=1000; i++)
    sum+=a*i;
```

Другая форма записи этого цикла содержит всего одно умножение, вместо 1000, поэтому более экономна:

```
sum=0;
for (i=1; i<=1000; i++)
    sum+=i;
sum*=a;
```

Внутри цикла могут встречаться выражения, фрагменты которых никак не зависят от переменной-счетчика этого цикла.

```
for (i=1; i<=m; i++)
    for (j=1; j<=n; j++)
        for (k=1; k<=p; k++)
            x=a*i*j+k;
```

Здесь слагаемое  $a*i*j$  в выражении для вычисления  $x$  не зависит от переменной  $k$ , поэтому для уменьшения количества вычислений можно использовать вспомогательную переменную  $y$ .

```
for (i=1; i<=m; i++)
    for (j=1; j<=n; j++) {
        y=a*i*j;
        for (k=1; k<=p; k++)
            x=y+k;
    }
```

Иногда *производительность программы целиком зависит от того, как запрограммирован цикл*.

На практике, однако, чтобы не внести новые ошибки, для устранения которых потребуется время, *к оптимизации уже существующей и отлаженной программы следует прибегать, только если в этом есть реальная необходимость*.

## Операторы перехода

### Оператор *break*

Оператор `break` осуществляет принудительный выход из циклов `for`, `while` и `do-while`, аналогично выходу из оператора `switch`.

Когда внутри цикла встречается оператор `break`, выполнение цикла немедленно прекращается (без проверки каких-либо условий) и управление передается оператору, следующему за ним. Например:

```
int i;
for (i=1; i<10; i++) {
    if (i==5)
        break;
    printf("%d ", i);
}
printf("\nПоследнее значение i=%d\n", i);
```

Здесь на пятой итерации цикла `for` выполнение цикла прерывается. На экран будет выведено:

```
1 2 3 4
Последнее значение i=5
```

Если оператор `break` находится внутри вложенного цикла, то прекращается выполнение только этого внутреннего цикла.

## Оператор *continue*

Оператор `continue` прерывает текущую итерацию цикла и осуществляет переход к следующей итерации. При этом все операторы до конца тела цикла пропускаются.

В цикле `for` оператор `continue` вызывает выполнение операторов приращения и проверки условия цикла. В циклах `while` и `do-while` – передает управление операторам проверки условий цикла.

В предыдущем примере для цикла `for` заменим `break` на `continue`.

```
int i;
for (i=1; i<10; i++) {
    if (i==5)
        continue;
    printf("%d ", i);
}
printf("\nПоследнее значение i=%d\n", i);
```

Тогда на пятой итерации значение переменной `i=5` не будет распечатано:

```
1 2 3 4 6 7 8 9
Последнее значение i=10
```

Если `continue` находится внутри вложенного цикла, то прекращается выполнение итерации только этого внутреннего цикла.

***! Замечание:*** Операторы `break` и `continue` нарушают нормы структурного программирования, результаты их работы могут быть достигнуты другими средствами. Поэтому избегать их использования считается хорошим стилем программирования. С другой стороны, принудительное прерывание работы циклов может приводить к ускорению работы программы. Таким образом, операторы `break` и `continue` нужно применять только там, где это действительно необходимо.

## Оператор goto

Оператор `goto` осуществляет безусловный переход к метке. При этом в программе должен присутствовать оператор с присвоенной ему меткой:

```
goto Метка;  
...  
Метка: Оператор
```

Правила именования меток те же, что и для любых идентификаторов.

***! Замечание:*** Использование оператора `goto` следует *избегать*, поскольку оно нарушает нормы структурного программирования.

Пожалуй, единственный случай, когда *использование `goto` опытным программистом (!) может быть оправдано – это принудительный выход из вложенного набора циклов при возникновении ошибки*. Например:

```
while (...) {  
    for (...) {  
        for (...) {  
            Операторы  
            if (Ошибка)  
                goto label; /* переход к метке */  
        }  
        Операторы  
    }  
    Операторы  
}  
Операторы  
label: УстранениеОшибки;
```

## Переполнение и системно-зависимые константы

### Границы целочисленных типов

Поскольку в языке C размер памяти, которую занимает каждый из типов данных, зависит от реализации компилятора, библиотек и аппаратной части, то и границы диапазонов допустимых значений могут быть разными.

В заголовочном файле `<limits.h>` определены константы, описывающие размеры **целочисленных** типов данных независимо от конкретной реализации. Вот некоторые из них:

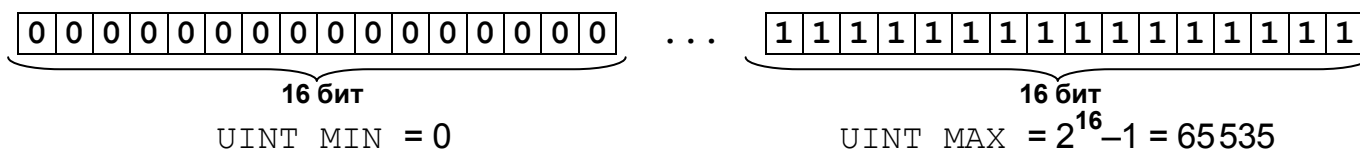
Константа	Описание
INT_MAX	максимальное значение <code>int</code>
INT_MIN	минимальное значение <code>int</code>
LONG_MAX	максимальное значение <code>long int</code>
LONG_MIN	минимальное значение <code>long int</code>
UINT_MAX	максимальное значение <code>unsigned int</code>
ULONG_MAX	максимальное значение <code>unsigned long int</code>

Данные константы удобно использовать в программах для отслеживания возможного **переполнения**.

## Переполнение

Рассмотрим упрощенную схему хранения переменной в памяти компьютера.

Если для хранения переменных типа `unsigned int` используется 2 байта, тогда говорят, что тип `int` является **16-битным** (в каждом байте – по 8 бит:  $8 \times 2 = 16$ ). Значит, всего с его помощью можно закодировать  $2^{16}$  различных значений. Для данного типа это – целые числа из диапазона  $[0; 2^{16}-1] = [0; 65\,535]$ , которые имеют двоичные коды:



В общем случае, если для переменной типа `unsigned int` используется  $n$  байт (т.е. тип **8× $n$ -битный**), всего можно закодировать  $2^{8n}$  различных значений из диапазона  $[0; 2^{8n}-1]$ , тогда: `UINT MIN = 0`, `UINT MAX =  $2^{8n}-1$` .

Пусть в программе объявлена переменная  $\bar{a}$  описанного 16-битного типа, значение которой задает пользователь. Допустим, значение этой переменной требуется увеличить на 10 000:

```
unsigned int a;  
printf("Введите число a: ");  
scanf("%d", &a);  
a += 10000;
```

Если пользователь введет значение `a` больше, чем 55535, то корректно выполнить операцию «сложение с присваиванием» (`+=`) будет невозможно – для хранения нового значения переменной 16 бит уже недостаточно.

При попытке выполнить сложение произойдет **переполнение** памяти, выделенной для хранения переменной `a`. В ней будет содержаться некорректное значение.

## Отслеживание переполнения

Чтобы избежать переполнения в описанном примере перед выполнением операции += необходимо провести проверку, не выйдет ли результат за границы диапазона возможных значений. Переполнения не произойдет, если выполняется неравенство:

$$a + 10\,000 \leq 65\,535.$$

где  $65535 = \text{UINT\_MAX}$

Казалось бы, в программе достаточно выполнить проверку:

```
if (a+10000 <= UINT_MAX)  a += 10000;    /* HEBEPHO! */
```

но здесь переполнение все равно произойдет при проверке условного выражения оператора `if` в операции сложения `a+10000`. Затем полученное некорректное значение будет сравниваться с константой `UINT_MAX`. Такая проверка была бы бессмысленна.

Поэтому, обычно **для проверки возможного переполнения при выполнении арифметической операции используют операцию, обратную к ней**. Проверочное неравенство можно представить так:

$$a \leq 65\,535 - 10\,000,$$

тогда код:

```
if (a <= UINT_MAX-10000) a += 10000;
```

не приведет к переполнению при проверке условного выражения.

Другой пример. Пусть для переменной типа `int`, значение которой введено пользователем, необходимо проверить, приведет ли к переполнению умножение этой переменной на 2:

```
#include <stdio.h>
#include <limits.h>
int main(void) {
    int n;
    printf("Макс.значение типа int = %d\n", INT_MAX);
    printf("Введите число n: ");
    scanf("%d", &n);
    if (n >= INT_MAX/2 + 1)
        printf("Увеличение числа n более, чем в 2 раза
                приведет к переполнению типа int.\n");
    return 0;
}
```

## Упражнения

### Упражнение 4.1

Составить программу, которая находит все простые числа в диапазоне от 1 до 1000. Простым называется натуральное число, которое делится только на 1 и на само себя.

### Упражнение 4.2

Составить программу, которая запрашивает у пользователя число  $n$ , и выводит на экран  $n$  строк в следующем виде:

```
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
. . .
```

### Упражнение 4.3

Составить программу, которая запрашивает у пользователя два числа типа `int` (в том числе отрицательные) и вычисляет их сумму (произведение) с предотвращением возможного переполнения.