НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Дисциплина «Информатика» (основной уровень), 1-й курс, 1-й семестр.

Методические указания

Тематическое занятие 3

Циклические алгоритмы.

Содержание

Инкремент, декремент и операции с присваиванием

Операции инкремента и декремента

Одноместная (унарная) операция *инкрементирования* добавляет к своему операнду единицу. Она обозначается символами `++` и может использоваться как в префиксной (`++a`), так и в постфиксной (`a++`) формах (здесь переменная `a` имеет целочисленный тип). Различие заключается в порядке выполнения операции: в выражении `++a` значение переменной увеличивается на `1` *до того*, как она используется, а в выражении `a++` – *после того*.

Например, пусть `a=5`, тогда следующие операторы дают разный результат

```
b = ++a; /* эквивалентно a=a+1; b=a; в результате a=6, b=6 */
```

```
b = a++; /* эквивалентно b=a; a=a+1; в результате a=6, b=5 */
```

Аналогично, одноместная операция *декрементирования* обозначается символами `--` и вычитает из своего операнда единицу. Например, `--a` или `a--`.

Эти две операции применимы только к переменным, выражения наподобие `(a+b)++` недопустимы.

В выражениях приоритет операций инкремента и декремента такой же высокий, как у унарных `+` и `-`. Порядок ассоциирования также справа налево (←).

Описанные свойства префиксной и постфиксной форм записи данных операций удобно использовать, например, при работе с индексами элементов массива. (Примеры будут приведены в соответствующих разделах.)

Но главное преимущество заключается в том, что использование операций инкремента и декремента позволяет уменьшить количество выполняемых программой операций присваивания (см. комментарий к примеру). А это приводит к увеличению скорости выполнения программы. (*Количество операций присваивания* – один из ключевых показателей эффективности работы компьютерных программ.)

Побочные эффекты

Для операций *инкремента* и *декремента* необходимо соблюдать правило: ***не следует применять эти операции к переменной, которая входит в выражение более одного раза***. Например, не следует использовать такое выражение:

```
b = a + 2 * a++;
```

Весьма показателен следующий пример. Для выражения:

```
b = (3 * a++) + (2 * a++);
```

не понятно, в какой момент значение переменной `a` будет увеличено на `1` (когда это произойдет в первый раз, и когда – во второй). Данный вопрос не оговорен в стандарте языка С, и целиком зависит от разработчиков используемой версии компилятора. Результат выполнения такой операции не определен, он называется ***«побочным эффектом»***. Подобных выражений необходимо тщательно избегать.

Операции с присваиванием

Выражения с операцией присваивания, в которых переменная из левой части повторяется в правой, можно записать в более компактной форме с помощью *операций с присваиванием*. Например:

`a = a + 3` можно записать так `a += 3`

Аналогично, для других операций с присваиванием запись вида

`ИмяПерем Опер= Выраж`

эквивалентна следующей записи

`ИмяПерем = (ИмяПерем) Опер (Выраж)`

здесь `Опер` – знак одной из операций: `+`, `-`, `*`, `/`, `%`. В этой записи следует обращать внимание на скобки – например, оператор

`a *= b + 2;` эквивалентен `a = a * (b + 2);`

Операции с присваиванием имеют такой же низкий приоритет, как и операция присваивания (=). Порядок ассоциирования тоже слева направо (→).

Помимо краткости записи, выражения с такими операциями более естественны для человеческого восприятия. Мы говорим «увеличить `a` на `3`», а не «извлечь `a`, прибавить `3` и поместить результат обратно в `a`».

Но основное преимущество в том, что операции с присваиванием выполняются быстрее, чем их эквивалентная форма. При их использовании компилятор генерирует оптимальный код, что способствует увеличению скорости выполнения программ.

Приоритет операций и порядок ассоциирования

Сводная таблица всех операций в языке С в порядке приоритета:

Операции	***Ассоциирование***	***Приоритет***
() [] -> .	→ слева направо	высокий (15)
! ~ ++ -- + - * & (тип**) sizeof**	← **справа налево**	(14)
*** / %**	→ слева направо	(13)
+ -	→ слева направо	(12)
<< >>	→ слева направо	(11)
< <= > >=	→ слева направо	(10)
== !=	→ слева направо	(9)
&	→ слева направо	(8)
^	→ слева направо	(7)
|	→ слева направо	(6)
&&	→ *слева направо*	(5)
||	→ *слева направо*	(4)
?:	← **справа налево**	(3)
= += -= *= /= %= &= ^= |= <<= >>=	← **справа налево**	(2)
,	→ слева направо	низкий (1)

Операторы цикла

Виды циклических конструкций

Цикл предназначен для выполнения повторяющихся действий. Цикл содержит группу операторов (*тело цикла*), выполнение которых повторяется необходимое количество раз. Каждое такое повторение называется *итерацией* цикла.

Существуют циклы с явно заданным числом повторений (с *параметром* или *счетчиком*). И циклы с неизвестным числом повторений (*итеративные*), которые проводят проверку заданного *условия* до или после выполнения тела цикла. Последние называются циклы с *предусловием* и циклы с *постусловием*, соответственно.

Цикл с предусловием `while`

Синтаксис оператора с предусловием содержит ключевое слово `while` (что означает «**пока**»), выражение условия и выполняемый оператор :

```
while (Выражение)
   Оператор
```

он похож на сокращенную форму оператора условия `if`. Результат условного `Выражения` принимает числовое значение.

Оператор цикла `while` (так же, как оператор `if`) вначале проверяет значение `Выражения`. Если оно истинно (принимает ***ненулевое*** значение), то выполняется тело цикла, содержащее один `Оператор`. Затем проверка `Выражения` повторяется еще раз. Так продолжается до тех пор, пока `Выражения` не станет ложным (примет нулевое значение).

Если в теле цикла требуется использовать несколько операторов:

```
while (Выражение) {
   Оператор1
   Оператор2
   ...
   ОператорN
}
```

! Замечание: ***Перед вызовом*** *любого оператора цикла необходимо провести подготовку – назначить* ***начальные значения*** *всем переменным, которые изменяются в теле цикла.*

Число повторений (итераций) оператора `while`, как правило, заранее неизвестно.

Для завершения цикла в его теле **обязательно** должны содержаться операторы, оказывающие **влияние на истинность** условного `Выражения`. Иначе цикл будет выполняться бесконечно – произойдет *зацикливание* программы.

Предотвратить зацикливание позволяет правило для итеративных циклов: ***переменная, которая участвует в*** `Выражении`***, обязательно должна изменяться в теле цикла.***

Пример. Суммирование целых чисел от 1 до 20:

```
#include <stdio.h>
int main(void) {
   int i, sum;  /* i – текущее число, sum – сумма */
   sum = 0;
   i = 1;  /* установка начальных значений */
   while (i<=20) {  /* проверка условного выражения цикла */
      sum += i;  /* накапливается сумма (sum=sum+i) */
      i++;  /* переход к следующему числу, и изменение условия */
   }
   printf("Сумма чисел от 1 до 20 равна %d\n", sum);
   return 0;
}
```

Цикл с постусловием `do-while`

Синтаксис оператора с постусловием содержит ключевые слова `do` и `while` (что означает «**выполнить**» и «**пока**»):

```
do {
   Оператор1
   Оператор2
   ...
   ОператорN
} while (Выражение);
```

Если в теле цикла содержится только один оператор, то фигурные скобки ставить не обязательно. Однако чтобы визуально не перепутать окончание цикла `do-while` с началом цикла с предусловием `while`, фигурные скобки желательно ставить всегда.

Оператор `do-while` вначале выполняет тело цикла, а затем проверяет значение `Выражения`. Далее аналогично циклу `while`. Если `Выражение` истинно (принимает ***ненулевое*** значение), то тело цикла повторяется еще раз. Так продолжается до тех пор, пока `Выражение` не станет ложным (примет нулевое значение).

Число итераций цикла `do-while`, как правило, также заранее не известно. Но тело цикла обязательно будет выполнено **хотя бы один раз.** А приведенное выше правило для итеративных циклов позволяет избежать зацикливания.

Тот же пример суммирования чисел:

```
#include <stdio.h>
int main(void) {
   int i, sum;  /* i - текущее число, sum - сумма */
   sum = 0;
   i = 1;  /* установка начальных значений */
   do {
      sum += i;  /* накапливается сумма (sum=sum+i) */
      i++;  /* переход к следующему числу, и изменение условия */
   } while (i<=20);  /* проверка условного выражения цикла */
   printf("Сумма чисел от 1 до 20 равна %d\n", sum);
   return 0;
}
```

Цикл со счетчиком `for`

В операторе цикла `for` может использоваться *переменная-параметр* (*счетчик*), которая на каждой итерации цикла изменяет свое значение согласно заданным правилам.

Синтаксис оператора содержит ключевое слово `for` (что означает «**для**»):

```
for (ВыражИниц; ВыражУсл; ВыражИзмен)
   Оператор
```

Эта конструкция эквивалентна следующей:

```
ВыражИниц;
while (ВыражУсл) {
   Оператор;
   ВыражИзмен;
}
```

В общем виде принцип работы цикла `for` следующий. Вначале один раз выполняется `ВыражИниц` – инициализация счетчика цикла, т.е. выражение с присваиванием начального значения некоторой переменной-параметру цикла. Затем проводится проверка `ВыражУсл` – условного выражения цикла. Если оно истинно (принимает ***ненулевое*** значение), то выполняется `Оператор` тела цикла. В конце итерации проводится изменение счетчика цикла согласно выражению `ВыражИзмен` (например, приращение счетчика на заданную

величину). После этого выполняется следующая итерация цикла `for`, т.е. следующая проверка условного выражения.

Так продолжается до тех пор, пока `ВыражУсл` не станет ложным (примет нулевое значение). Следует заметить, что после этой последней проверки изменение счетчика не произойдет, а цикл сразу окончится.

Если в теле цикла требуется использовать несколько операторов:

```
for (ВыражИниц; ВыражУсл; ВыражИзмен) {
   Оператор1
   Оператор2
   ...
   ОператорN
}
```

Как правило, оператор `for` используется для организации циклов с числом повторений, которое известно заранее, к моменту вызова цикла. Желательно использовать переменную-параметр (счетчик) целочисленного типа.

Тот же пример суммирования чисел:

```
#include <stdio.h>
int main(void) {
   int i, sum;  /* i – текущее число (счетчик), sum – сумма */
   sum = 0;  /* установка начального значения суммы */
   for (i=1; i<=20; i++)  /* управление циклом */
      sum += i;  /* накапливается сумма (sum=sum+i) */
   printf("Сумма чисел от 1 до 20 равна %d\n", sum);
   return 0;
}
```

Значение счетчика цикла может быть как угодно изменено в теле самого цикла. Причем при выходе из цикла счетчик сохраняет свое значение.

```
int i, k=0;
for (i=1; i<=10; i+=2) {
   i += 3;  /* такое действие допускается */
   printf("k=%d i=%d\n", ++k, i);
}   /* две итерации: «k=1 i=4» и «k=2 i=9» */
printf("Result: i=%d\n", i);  /* результат: «i=11» */
```

Зацикливание (бесконечный цикл)

Любое из трех выражений цикла `for` можно опустить, при этом точки с запятыми должны оставаться на своих местах. Если опустить выражения `ВыражИниц` и `ВыражИзмен`, то соответствующие операции не будут выполняться. Если же опустить проверку условия `ВыражУсл`, то по умолчанию считается, что условие продолжения цикла всегда истинно, и такая конструкция станет *бесконечным циклом* (*зациклится*):

```
for ( ; ; ) {      /* зацикливание */
   ...
}
```

Такой цикл должен прерываться другими способами, например с помощью оператора `break`.

Сравнение операторов цикла

Большинство задач можно решить, используя любой из операторов цикла, но в некоторых случаях предпочтительнее использовать один из них.

Самым универсальным из операторов цикла является цикл с предусловием `while`, т.к. операторы в его теле могут быть не выполнены ни разу. Доказано, что любая программа может быть написана, используя только итеративную конструкцию `while` и оператор условия `if`.

Цикл `for` следует предпочесть тогда, когда есть простая инициализация и простые правила изменения счетчика цикла, поскольку в этом случае все управляющие элементы цикла удобно сгруппированы вместе в его заголовке.

Обобщение рассмотренного примера суммирования чисел от 1 до 20:

```
sum = 0;
i = 1;
while (i<=20) {
   sum += i;
   i++;
}
```

```
sum = 0;
i =1;
do {
   sum += i;
   i++;
} while (i<=20);
```

```
sum:=0;
for (i=1; i<=20; i++)
   sum += i;
```

Примеры

Вычисление факториала n!

```
#include <stdio.h>
#include <locale.h>
int main(void) {
   int      n, i;  /* n - число, i - счетчик */
   long int fact;  /* fact - значение факториала */
   setlocale(LC_ALL,"");
   printf("Введите число n: ");   scanf("%d", &n);
   fact = 1;  /* установка начального значения факториала */
   for (i=1; i<=n; i++)  /* управление циклом */
      fact *= i;  /* накапливается значение (fact=fact*i) */
   printf("Факториал n!=%d\n", fact);
   return 0;
}
```

Далее в примерах при использовании национальных стандартов будем опускать подключение заголовочного файла `<locale.h>` и вызов функции `setlocale()`.

Вычисление НОД двух чисел

Наибольший общий делитель (НОД) двух чисел – это наибольшее целое число, на которое оба числа делятся без остатка.

Алгоритм Евклида основан на следующем свойстве целых чисел. Пусть целые неотрицательные числа a и b одновременно не равны нулю и $a \geq b$. Тогда, если $b=0$, то $НОД(a,b)=a$. Если $b \neq 0$, то для чисел a, b и r, где r – остаток от деления a на b, выполняется равенство $НОД(a,b)=НОД(b,r)$. Действительно, $r = a \bmod b = a - (a \operatorname{div} b) \cdot b$, где *mod* и *div* – остаток и частное от целочисленного деления соответственно. Если какое-то число делит нацело и a и b, то из приведенного равенства следует, что оно делит нацело и числа r и b.

Реализация алгоритма Евклида:

```
#include <stdio.h>
int main(void) {
   long int a, b;
   printf("Введите два числа, не равные нулю: ");
   scanf("%d %d", &a, &b);
   do {
      if (a>b)  a %= b;
      else      b %= a;
   } while (a!=0 && b!=0);
   printf("НОД =%d\n", a+b);
   return 0;
}
```

Упражнения

Упражнение 3.1

Составить программу, которая с помощью цикла определяет, является ли простым число, введенное пользователем. Простым называется натуральное число, которое делится только на 1 и на само себя.

Упражнение 3.2

Составить программу, которая запрашивает у пользователя целые числа (до тех пор, пока пользователь не введет число 0) и вычисляет их сумму.

Упражнение 3.3

Составить программу, которая вычисляет степень числа n^k, натуральные числа n и k вводятся пользователем.