Тема 1.2 Операторы организации циклов. Простой и составной оператор.

Обычно операторы программы выполняются друг за другом в той последовательности, в которой они написаны. Это называется последовательным выполнением. Однако, различают операторы в С++ позволяющие программисту указать, то следующим должен выполняться не очередной оператор в тексте программы, а какой-то другой. Это называется передачей управления. Поэтому было доказано, то все программы могут быть написаны с использованием всего трех управляющих структур, названных структура следования, выбора и повторения.

Структура следования встроена в С++. Пока не указано иное, компьютер выполняет операторы С++ один за другим.

С++ обеспечивает три типа **структур выбора** альтернатив:

- структура выбора IF;
- структура IF/ELSE;
- структура SWITCH.

Структура выбора IF (если) называется структурой с единственным выбором, поскольку она выбирает или игнорирует единственное действие.

Формат оператора:

if (выражение) оператор-1;

Пример:

```
…
if (i <=60)    i++;
…
```

Структура IF/ELSE (если/ иначе) называется структурой с двойным выбором, так как она осуществляет выбор между двумя различными действиями.

Формат оператора:

if (выражение) оператор-1; [else оператор-2;]

Выполнение оператора if начинается с вычисления выражения.

Далее выполнение осуществляется по следующей схеме:

- если выражение истинно (т.е. отлично от 0), то выполняется оператор-1.
- если выражение ложно (т.е. равно 0),то выполняется оператор-2.
- если выражение ложно и отсутствует оператор-2 (в квадратные скобки заключена необязательная конструкция), то выполняется следующий за if оператор.

После выполнения оператора if значение передается на следующий оператор программы, если последовательность выполнения операторов программы не будет принудительно нарушена использованием операторов перехода.

Пример:

```
…
if (i < j)    i++:
else
{ j = i-3;
i++;
}
…
```

Рассмотрим, как написать программу для вычисления значений x, y, z по формулам:

$Y = x^3$-6.5, если x<=9 и z = y, если y<8

$\sqrt{x}$+8.7 + 5.2, если x>9 y^2, если y=8

Y^3, если y>8

```
#include <iostream.h>
#include <math.h>

void main ()
{
        double x,y,z;
        cout<<"\n Input x: \n";
        cin>>x;

        if (x<=9) y=pow(x, 3) – 6.5;
                else y=sqrt(x+8.7) + 5.2;

        cout<<"y = "<<y <<"\n";

        if (y<8) z = y;
                else if (y==8) z=pow(y, 2);
                        else z=pow(y, 3);

        cout<<"z = "<<z<<"\n";
}
/* Результат выполнения программы:
Input x:
2
y = 1.5
z = 1.5
Press any key to continue
*/
```

Допускается использование вложенных операторов if. Оператор if может быть включен в конструкцию if или в конструкцию else другого оператора if. Чтобы сделать программу более читабельной, рекомендуется группировать операторы и конструкции во вложенных операторах if, используя фигурные скобки. Если же фигурные скобки опущены, то компилятор связывает каждое ключевое слово else с наиболее близким if, для которого нет else.

Примеры:

```
int main ( )
 {
  int t=2, b=7, r=3;
    if (t>b)
     {
        if (b < r)  r=b;
     }
    else r=t;
  return (0);
 }
```

В результате выполнения этой программы r станет равным 2.

Если же в программе опустить фигурные скобки, стоящие после оператора if, то программа будет иметь следующий вид:

```
int main  ( )
 {
  int t=2,b=7,r=3;
  if ( a>b )
     if ( b < c ) t=b;
     else       r=t;
  return (0);
 }
```

В этом случае r получит значение равное 3, так как ключевое слово else относится ко второму оператору if, который не выполняется, поскольку не выполняется условие, проверяемое в первом операторе if.

Структура SWITCH (выбор) называется структурой с множественным выбором, так как она осуществляет выбор среди множества различных действий.

```
switch ( выражение )
   {   [объявление]
          :
       [ case  константное-выражение1]: [ список-операторов1]
       [ case  константное-выражение2]: [ список-операторов2]
          :
          :
       [ default: [ список операторов ]]
   }
```

Выражение, следующее за ключевым словом switch в круглых скобках, может быть любым выражением, допустимыми в языке С, значение которого должно быть целым.

Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора smitch состоит из нескольких операторов, помеченных ключевым словом case с последующим константным-выражением. Следует отметить, что использование целого константного выражения является существенным недостатком, присущим рассмотренному оператору.

Так как константное выражение вычисляется во время трансляции, оно не может содержать переменные или вызовы функций. Обычно в качестве константного выражения используются целые или символьные константы.

Все константные выражения в операторе switch должны быть уникальны. Кроме операторов, помеченных ключевым словом case, может быть, но обязательно один, фрагмент помеченный ключевым словом default.

Список операторов может быть пустым, либо содержать один или более операторов. Причем в операторе switch не требуется заключать последовательность операторов в фигурные скобки.

Отметим также, что в операторе switch можно использовать свои локальные переменные, объявления которых находятся перед первым ключевым словом case, однако в объявлениях не должна использоваться инициализация.

Схема выполнения оператора switch следующая:

- вычисляется выражение в круглых скобках;
- вычисленные значения последовательно сравниваются с константными выражениями, следующими за ключевыми словами case;
- если одно из константных выражений совпадает со значением выражения, то управление передается на оператор, помеченный соответствующим ключевым словом case;
- если ни одно из константных выражений не равно выражению, то управление передается на оператор, помеченный ключевым словом default, а в случае его отсутствия управление передается на следующий после switch оператор.

Отметим интересную особенность использования оператора switch: конструкция со словом default может быть не последней в теле оператора switch. Ключевые слова case и default в теле оператора switch существенны только при начальной проверке, когда определяется начальная точка выполнения тела оператора switch. Все операторы, между начальным оператором и концом тела, выполняются вне зависимости от ключевых слов, если только какой-то из операторов не передаст управления из тела оператора switch. Таким образом, программист должен сам позаботится о выходе из case, если это необходимо. Чаще всего для этого используется оператор break.

Для того, чтобы выполнить одни и те же действия для различных значений выражения, можно пометить один и тот же оператор несколькими ключевыми словами case.

Пример:

```
int i=2;
switch (i)
  {
   case 1: i += 2;
   case 2: i *= 3;
   case 0: i /= 2;
   case 4: i -= 5;
   default:     ;
}
```

Выполнение оператора switch начинается с оператора, помеченного case 2. Таким образом, переменная i получает значение, равное 6, далее выполняется оператор, помеченный ключевым словом case 0, а затем case 4, переменная i примет значение 3, а затем значение -2. Оператор, помеченный ключевым словом default, не изменяет значения переменной.

Отметим, что в теле оператора switch можно использовать вложенные операторы switch, при этом в ключевых словах case можно использовать одинаковые константные выражения.

Пример:

```
 :
switch (a)
 {
  case 1: b=c; break;
  case 2:
     switch (d)
     {  case 0:  f=s;  break;
        case 1:  f=9;  break;
        case 2:  f-=9; break;
     }
   case 3: b-=c; break;
   :
 }
```

Операторы BREAK, CONTINUE

Часто бывает необходимо перейти на следующую итерацию цикла еще до завершения выполнения всех операторов тела цикла. Для этого используется оператор continue.

Кроме того, в ряде случаев требуется выйти за пределы цикла, даже если условия продолжения цикла выполняются. В этом случае используется оператор break.

Пример использования этих операторов приведен в листинге 4.9. Это несколько усложненный вариант уже знакомой игры. В этом случае, кроме меньшего и большего значений, предлагается ввести шаг и целевое значение. Ya каждой итерации цикла значение переменной small увеличивается на единицу. Значение large уменьшается на два, если меньшее число не кратно значению переменной шага (skip). Игра заканчивается, когда значение переменой small становится больше, чем значение large. Если значение переменной large совпадает с целевым значением (target), выводится сообщение и игра прерывается.

Цель игры состоит в том, чтобы угадать число, в которое "попадет" значение target.

```
Листинг 4.9. Использование break и continue
1: // Листинг 4.9.
2: // Пример использования операторов break и continue
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:    unsigned short small;
9:    unsigned long large;
10:   unsigned long skip;
```

```
11:    unsigned long target;
12:    const unsigned short MAXSMALL=65535;
13:
14:    cout << "Enter a small number: ";
15:    cin >> small;
16:    cout << "Enter a large number: ";
17:    cin >> large;
18:    cout << "Enter a skip number: ";
19:    cin >> skip;
20:    cout << "Enter a target number: ";
21:    cin >> target;
22:
23:    cout << "\ n";
24:
25:    // установка условий продолжения цикла
26:    while (small < large && large > 0 && small < MAXSMALL)
27:
28:    {
29:
30:       small++;
31:
32:       if (small % skip == 0) // уменьшить значение large?
33:       {
34:          cout << "skipping on:" << small << endl;
35:          continue;
36:       }
37:
38:       if (large == target) // проверка попадания в цель
39:       {
40:          cout << " Target reached!";
41:          break;
42:       }
43:
44:       large-=2;
45:    } // конец цикла
46:
47:    cout << "\ nSmall: " << small << " Large: " << large <<
endl;
48:    return 0;
49: }
```

```
Результат:
Enter a small number: 2
Enter a large number: 20
Enter a skip number: 4
Enter a target number: 6
skipping on 4 skipping on 8
Small: 10
Large: 8
```

Как видим, игра закончилась поражением пользователя, поскольку меньшее значение превысило большее, а цель так и не была достигнута.

В строке проверяются условия продолжения цикла. Если значение переменной small меньше значения large, а также если large больше нуля и small не превышает значение константы SMALLINT, управление передается первому оператору тела цикла.

В строке 32 вычисляется остаток от деления значения переменной small на значение skip. Если значение small кратно skip, оператор continue запускает следующую итерацию цикла (срока

26). В результате такого перехода пропускается проверка целевого значения и операция уменьшения значения переменной large.

Сравнение значений target и large выполняется в строке 38. Если эти значения равны, игра заканчивается победой пользователя. В этом случае программа выводит сообщение о победе, работа цикла прерывается оператором break и управление передается в строку 46.

Оператор break обеспечивает прекращение выполнения самого внутреннего из объединяющих его операторов switch, do, for, while. После выполнения оператора break управление передается оператору, следующему за прерванным.

Оператор continue, как и оператор break, используется только внутри операторов цикла, но в отличие от него выполнение программы продолжается не с оператора, следующего за прерванным оператором, а с начала прерванного оператора.

Оператор continue, как и оператор break, прерывает самый внутренний из объемлющих его циклов.

Оператор goto

Для решения ряда задач часто требуется многократное выполнение одних и тех же действий. На практике это реализуется с помощью рекурсивных или итеративных алгоритмов. Суть итеративного процесса заключается в повторении последовательности операций нужное количество раз.

В те годы, когда программирование находилось еще на начальной стадии развития, использовались только небольшие по размеру и достаточно примитивные программы. Нельзя было назвать приятным и сам процесс их разработки. В таких программах циклы состояли из метки, последовательности команд и оператора безусловного перехода.

В C++ меткой называют идентификатор, за которым следует двоеточие (:). Метка всегда устанавливается перед оператором, на который необходимо будет передать управление. Для перехода на нужную метку используется оператор goto, за которым следует имя метки. Пример использования оператора goto приведен в листинге 4.1.

```
Листинг 4.1 Организация цикла с помощью оператора goto
1: // Листинг 4.1.
2: // Организация цикла с помощью goto
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:       int counter = 0; // инициализация счетчика
9: loop: counter ++; // начало цикла
10:      cout << "counter: " << counter << "\ n";
11:      if (counter < 5) // проверка значения
12:         goto loop; // возвращение к началу
13:
14:      cout << "Complete. Counter: " << counter << n";
15:      return 0;
16: }
Результат:

counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
Comlete. Counter: 5
```

В строке 8 переменная counter инициализируется нулевым значением. Метка loop: в строке 9 показывает начало цикла. На каждой итерации значение counter увеличивается на единицу и

выводится на экран. В строке 11 выполняется проверка значения переменной counter. Если оно меньше пяти, значит условие выполняется и управление передается оператору goto, в результате чего осуществляется переход на строку 9. Итеративный процесс выполняется до тех пор, пока значение переменной counter не достигнет пяти. После этого программа выходит за пределы цикла и на экран выводится окончательный результат.

Почему следует избегать оператора goto

Со временем нелестные высказывания в адрес оператора goto участились, впрочем, вполне заслуженно. С помощью оператора goto можно осуществлять переход в любую точку программы — вперед или назад. Такое беспорядочное использование этого оператора привело к появлению запутанных и абсолютно непригодных для восприятия программ, получивших жаргонное название "спагетти". Поэтому последние двадцать лет преподаватели программирования во всем мире твердили студентам одну и ту же фразу: "Никогда не используйте оператор goto".

На смену оператору goto пришли конструкции с несколько более сложной структурой, но и с более широкими возможностями: for, while и do. . .while. Несмотря на то что после полного искоренения оператора goto структура программ значительно прояснилась, негативные высказывания в его адрес следует признать преувеличенными. Как любой инструмент программирования, при правильном использовании оператор goto может оказаться достаточно полезным. В силу этого комитет ANSI принял решение оставить этот оператор в языке. Правда, вместе с этим родилась шутка: "Дети! Использование этого оператора в домашних условиях небезопасно!"

C++ обеспечивает три типа **структур повторения**, называемых:

- WHILE;
- DO/ WHILE;
- FOR.

Оператор цикла while называется циклом с предусловием и имеет следующий формат:

while (выражение) тело;

В качестве выражения допускается использовать любое выражение языка С, а в качестве тела любой оператор, в том числе пустой или составной.

Схема выполнения оператора while следующая:

1. Вычисляется выражение.
2. Если выражение ложно, то выполнение оператора while заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполняется тело оператора while.
3. Процесс повторяется с пункта 1.

В операторе while вначале происходит проверка условия. Поэтому оператор while удобно использовать в ситуациях, когда тело оператора не всегда нужно выполнять.

Внутри оператора while можно использовать локальные переменные, которые должны быть объявлены с определением соответствующих типов.

Листинг 4.2. Организация цикла с помощью оператора while

```
1: // Листинг 4.2.
2: // Организация цикла с помощью оператора while
3:
4: include <iostream.h>
5:
6: int main()
7: {
8:       int counter = 0; // присвоение начального значения
```

```
9:
10:      while(counter < 5) // проверка условия продолжения цикла
11:      {
12:          counter++; // тело цикла
13:          cout << " counter: " << counter << "\ n";
14:       }
15:
16:       cout << " Complete. Counter: " << counter << n";
17:       return 0;
18: }
```

```
Результат:
counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
Comlete. Counter: 5
```

Эта несложная программа показывает пример организации цикла с помощью оператора while. В начале каждой итерации проверяется условие, и, если оно выполняется, управление передается на первый оператор цикла. В нашем примере условию продолжения цикла удовлетворяют все значения переменной counter, меньшие пяти (строка 10). Если условие выполняется, запускается следующая итерация цикла. В строке 12 значение счетчика увеличивается на единицу, а в строке 13 выводится на экран. Как только значение счетчика достигает пяти, тело цикла (строки 11 — 14) пропускается и управление передается в строку 15.

Сложность логического выражения, являющегося условием в операторе while, не ограничена. Это позволяет использовать в конструкции while любые логические выражения C++. При построении выражений допускается использование логических операций: && (логическое И), || (логическое ИЛИ), а также ! (логическое отрицание). В листинге 4.3 показан пример использования более сложных условий в конструкциях с оператором while.

Листинг 4.3 Сложные условия в конструкциях while

```
1: // Листинг 4.3.
2: // Сложные условия в конструкциях while
3:
4: include <iostream.h>
5:
6: int main()
7: {
8:    unsigned short small;
9:    unsigned long large;
10:   const unsigned short MAXSMALL=65535;
11:
12:   cout << "Enter a small number: ";
13:   cin >> small;
14:   cout << "Enter a large number: ";
15:   cin >> large;
16:
17:   cout << "small: " << small << "...";
18:
19:   // на каждой итерации проверяются три условия
20:   while (small < large && large > 0 && small < MAXSMALL)
22:   {
23:       if (small % 5000 == 0) // после каждых 5000 строк
выводится точка
24:           cout << ".";
```

```
25:
26:       small++;
27:
28:       large-=2;
29:   }
30:
31:   cout << "\ nSmall: " << small << " Large: " << large <<
endl;
32:   return 0;
33: }
```

```
Результат:
Enter a small number: 2 Enter a large number: 100000
small: 2........
Small: 33335
Large: 33334
```

Программа представляет собой простую логическую игру. Вначале предлагается ввести два числа — small и large. После этого меньшее значение увеличивается на единицу, а большее уменьшается на два до тех пор, пока они не "встретятся". Цель игры: угадать число, на котором значения "встретятся".

В строках 12—15 осуществляется ввод значений. В строке 20 проверяется три условия продолжения цикла.

1) Значение переменной small не превышает значения large.

2) Значение переменной large неотрицательное и не равно нулю.

3) Значение переменной small не превышает значения константы MAXSMALL.

Далее, в строке 23, вычисляется остаток от деления числа small на 5000, причем значение переменной small не изменяется. Если small делится на 5000 без остатка, результатом выполнения этой операции будет 0. В этом случае для визуального представления процесса вычислений на экран выводится точка. Затем в строке 26 значение переменной small увеличивается на 1, а в строке 28 значение large уменьшается на 2.

Цикл завершается, если хотя бы одно из условий перестает выполняться. После этого управление передается в строку 29, следующую за телом цикла.

Оператор цикла do while называется оператором цикла с постусловием и используется в тех случаях, когда необходимо выполнить тело цикла хотя бы один раз.

Формат оператора имеет следующий вид:

do тело while (выражение);

Схема выполнения оператора do while :

1. Выполняется тело цикла (которое может быть составным оператором).

2. Вычисляется выражение.

3. Если выражение ложно, то выполнение оператора do while заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполнение оператора продолжается с пункта 1.

Пример 1:

```
//считать до 10
int x = 0;
do
    cout << "X: " << x++;
while (x < 10);
```

При организации циклов с помощью оператора while возможна ситуация, когда тело цикла вообще не будет выполняться. Поскольку условие продолжения цикла проверяется в начале каждой итерации, при нарушении истинности выражения, задающего это условие, выполнение цикла будет прервано еще до запуска первого оператора тела цикла. Пример такой ситуации приведен в листинге 4.4.

Листинг 4.4 Преждевременное завершение цикла с while

```
1: // Листинг 4.4.
2: // Если условие продолжения цикла не выполняется,
3: // тело цикла пропускается.
4:
5: #include <iostream.h>
6:
7: int main()
8: {
9:     int counter;
10:    cout « "How many hellos?: ";
11:    cin >> counter;
12:    while (counter > 0)
13:    {
14:        cout « "Hello!\ n";
15:        counter--;
16:    }
17:    cout << "Counter is OutPut: " << counter;
18:    return 0;
19: }
```

```
Результат:
How many hellos?: 2
Hello!
Hello!
Counter is OutPut: 0
How many hellos?: 0
Counter is OutPut: 0
```

В строке 10 вам предлагается ввести начальное значение счетчика, которое записывается в переменную counter. В строке 12 это значение проверяется, а затем в теле цикла уменьшается на единицу. При первом запуске программы начальное значение счетчика равнялось двум, поэтому тело цикла выполнялось дважды. Во втором случае было введено число 0. Понятно, что в этом случае условие продолжения цикла не выполнялось и тело цикла было пропущено. В результате приветствие не было выведено ни разу.

Как же поступить, чтобы сообщение выводилось по крайней мере один раз?

При использовании конструкции do. . . while условие проверяется после выполнения тела цикла. Это гарантирует выполнение операторов цикла по крайней мере один раз. В листинге 4.5 приведен измененный вариант предыдущей программы, в котором вместо оператора while используется конструкция do.. while.

Листинг 4.5. Использование конструкции do...while

```
1: // Листинг 4.5.
2: // Пример использования конструкции do...while
3:
4: include <iostream.h>
5:
6: int main()
7: {
8:     int counter;
9:     cout << "How many hellos? ";
```

```
10:     cin >> counter;
11:     do
12:     {
13:          cout « "Hello\ n";
14:          counter--;
15:     } while (counter >0 );
16:     cout << "Counter is: " << counter << endl;
17:     return 0;
18: }
Результат:

How many hellos?: 2
Hello
Hello
Counter is : 0
```

В строке 9 пользователю предлагается ввести начальное значение счетчика, которое записывается в переменную counter. В конструкции do.. .while условие проверяется в конце каждой итерации, что гарантирует выполнение тела цикла по меньшей мере один раз. В строке 13 на экран выводится текст приветствия, а в строке 14 значение переменной counter уменьшается на единицу. Условие продолжения цикла проверяется в строке 15. Если оно истинно, выполняется следующая итерация цикла (строка 13). В противном случае цикл завершается и управление передается в строку 16.

Чтобы прервать выполнение цикла до того, как условие станет ложным, можно использовать оператор break.

Операторы while и do while могут быть вложенными.

Пример:

```
int i,j,k;
...
i=0;  j=0;  k=0;
do { i++;
    j--;
    while (k < i)  k++;
   }
while (i<30 && j<-30);
```

Оператор for - это наиболее общий способ организации цикла.

Он имеет следующий формат:

for (выражение 1 ; выражение 2 ; выражение 3) тело

Выражение 1 обычно используется для установления начального значения переменных, управляющих циклом. Выражение 2 - это выражение, определяющее условие, при котором тело цикла будет выполняться. Выражение 3 определяет изменение переменных, управляющих циклом после каждого выполнения тела цикла.

Схема выполнения оператора for:

1. Вычисляется выражение 1.
2. Вычисляется выражение 2.
3. Если значения выражения 2 отлично от нуля (истина), выполняется тело цикла, вычисляется выражение 3 и осуществляется переход к пункту 2, если выражение 2 равно нулю (ложь), то управление передается на оператор, следующий за оператором for.

Существенно то, что проверка условия всегда выполняется в начале цикла. Это значит, что тело цикла может ни разу не выполниться, если условие выполнения сразу будет ложным.

Пример:

```
int main()
{ int i,b;
  for (i=1; i<10; i++) b="i*i;" return 0; }
```

В этом примере вычисляются квадраты чисел от 1 до 9.

Оператор цикла for может быть заменен оператором while следующим образом:

```
выражение-1;
while (выражение-2)
{ тело
  выражение-3;
}
```

Множественные инициализации и приращения счетчиков цикла

Синтаксис задания цикла for позволяет инициализировать несколько переменных-счетчиков, проверять сложные условия продолжения цикла или последовательно выполнять несколько операций над счетчиками цикла. Если присваиваются значения нескольким счетчикам или выполняется несколько операций, они записываются последовательно и разделяются запятыми. В листинге 4.7 инициализируются два счетчика, значения которых после каждой итерации увеличиваются на единицу.

Листинг 4.7 Использование нескольких счетчиков в цикле for

```
1: //Листинг 4.7.
2: // Использование нескольких счетчиков
3: // в цикле for
4:
5: #include <iostream.h>
6:
7: int main()
8: {
9:    for (int i=0, j=0; i<3; i++, j++)
10:      cout << "i: " << i << " j: " << j << endl;
11:   return 0;
12: }
```

Результат:

```
i: 0 j: 0

i: 1 j: 1
i: 2 j: 2
```

В строке 9 переменные i и j инициализируются нулевыми значениями. Затем проверяется условие i<3 и, так как оно справедливо, выполняется первая итерация цикла. На каждой итерации осуществляется вывод значений счетчиков на экран. После этого выполняется третья часть конструкции for, в которой значения переменных-счетчиков увеличиваются на единицу. После выполнения строки 10 и изменения значений переменных условие проверяется снова. Если условие все еще справедливо, запускается следующая итерация цикла. Это происходит до тех пор, пока условие продолжения цикла не нарушится. В этом случае значения переменных не изменяются и управление передается следующему после цикла оператору.

Нулевые параметры цикла for

Любой параметр цикла for может быть опущен. Пропуск означает использование так называемого нулевого параметра. Нулевой параметр, как и любой другой, отделяется от остальных параметров цикла for символом точки с запятой (;). Если опустить первый и третий параметры цикла for, как показано в листинге 4.8, результат его применения будет аналогичен полученному при использовании оператора while.

Листинг 4.8. Нулевые параметры цикла for

```
1: // Листинг 4.8.
2: // Нулевые параметры цикла for
3:
4: #include <iostream.h>
5:
6: int main()
7: {
8:    int counter = 0;
9:
10:   for( ; counter < 5; )
11:   {
12:       counter++;
13:       cout << "Looping! ";
14:   }
15:
16:   cout << "\ nCounter: " << counter << "\ n";
17:   return 0;
18: }
```

Результат:

```
Looping! Looping! Looping! Looping! Looping!
Counter: 5.
```

В строке 8 присваивается значение переменной counter. Установки параметров цикла for, показанные в строке 10, содержат только проверку условия продолжения цикла. Операция над переменной цикла в конструкции for также опущена. Таким образом, этот цикл можно представить в виде:

```
while (counter < 5).
```

Рассмотренный пример еще раз показывает, что возможности языка C++ позволяют решить одну и ту же задачу множеством способов. Листинг 4.8 приведен скорее для иллюстрации гибкости возможностей C++, поскольку ни один опытный программист не будет использовать цикл for подобным образом. Тем не менее можно опустить даже все три параметра цикла for, а для управления циклом использовать операторы break и continue.

Область видимости переменных-счетчиков циклов for

До недавнего времени область видимости переменных, описанных в цикле for, распространялась на весь текущий блок. Согласно новому стандарту, установленному ANSI, область видимости переменных, описанных в таком цикле, должна распространяться только на тело цикла. Следует заметить, что, несмотря на внесенные изменения, многие компиляторы продолжают поддерживать только старый стандарт. Набрав приведенный ниже фрагмент программного кода, можно проверить свой компилятор на соответствие новому стандарту.

```
#include <iostream. h>
int main()
{
   // Проверка области видимости переменной i
```

```
        for (int i = 0; i<5; i++)
        {
           cout « "i: " << i << endl;
        }
        i = 7; // i находится за пределами области видимости
        return 0;
     }
```

Если такая программа будет компилироваться без ошибок, значит, ваш компилятор еще не поддерживает нового стандарта ANSI.

Компиляторы, соответствующие новому стандарту, должны сгенерировать сообщение об ошибке для выражения i = 7. После внесения некоторых изменений программа будет восприниматься всеми компиляторами без ошибок.

```
     #include <iostream.h>
     int main()
     {
        int i; //объявление переменной за пределами цикла
        for (int i = 0; i<5; i++)
        {
           cout << "i: " << i << endl;
        }
        i = 7; // теперь переменная i будет корректно восприниматься
всеми компиляторами
        return 0;
     }
```

C++ имеет только семь управляющих структур: следования, три типа повторения и три типа выбора. Следовательно любая программа на C++ формируется из такого количества комбинаций каждого типа управляющих структур, которое нужно для осуществления соответствующего алгоритма.

Тема 1.2 Итерационные алгоритмы и программы

Итерация – циклическая управляющая структура, которая содержит композицию и ветвление, предназначенное для организации повторяющихся процессов обработки последовательности значений переменных.

Итерационными (пошаговыми) алгоритмами называются алгоритмы, в которых на каждом шаге используется одна и та же формула, выраженная через значения, полученные на предыдущих шагах алгоритма.

В итерационных циклах на каждом шаге вычислений происходит последовательное приближение и проверка условия достижения искомого результата. Выход из итерационного цикла осуществляется в случае выполнения заданного условия. Различают итерационные циклы с предусловиями и с постусловиями.

Итерационный процесс – процесс последовательного вычисления значений по формулам; процесс последовательных приближений.

Итерационный цикл

Итак, любой цикл как процесс можно охарактеризовать четырьмя составляющими:

- начальное состояние;
- ограничение цикла (условие продолжения или завершения):
- переход к следующему шагу:
- повторяющееся действие – тело цикла.

В большинстве циклов действия, производимые в теле цикла, не влияют на параметры его протекания: количество шагов, характеристики шага. В таких «хороших» циклах параметры заголовка цикла не зависят от значений переменных, вычисляемых в теле цикла, и цикл имеет постоянное количество повторений, например:

```
for (i=0; i<n; i++) { ...A[i]... }
```

Если же поведение программы на некотором шаге цикла может зависеть от результатов выполнения тела цикла на предыдуших шагах, либо число повторений цикла зависит от результатов выполнения шага, такие циклы и программируемые ими процессы называются итерационными. Наиболее широко они применяются в вычислительной математике, когда для получения численного результата используется итерационный цикл последовательных приближений к нему.

Если изобразить общую схему итерационного цикла, то в нем обязательно будут переменные, сохраняющие результат предыдущего (x1) и еще более ранних (x2,...) шагов, а также переменная x - результат текущего шага:

```
for (x1=...,x2=...; условие(x1,x2); x2=x1,x1=x)  { ...x =…x1…x2…;...}
```

Образно говоря, если x – «сегодняшнее» значение, то x1 – вчерашнее, а x2 – позавчерашнее. При переходе к следующему шагу «вчерашнее» становится

«позавчерашним», «сегодняшнее» – «вчерашним». Особенность итерационного цикла состоит в том, что

Если в итерационном цикле гарантируется выполнение одного шага, то может быть использован цикл do...while.

```
x=...; x1=...;                          // Начальное значение текущего шага
do      {
x2 = x1; x1 = x;                  // Следующий шаг
x = …x1…x2…;                    // Результат текущего шага
} while (условие(x2,x1,x));        // Условие завершения
```

Если использовать результат только текущего шага, который зависит от результата предыдущего, то схему цикла можно упростить.

```
for (x=...; условие(x);  ) { ...x = …x…;... }
```

Итерационные алгоритмы используются при реализации итерационных численных методов. В итерационных алгоритмах необходимо обеспечить обязательное достижение условия выхода из цикла (сходимость итерационного процесса). В противном случае произойдет зацикливание алгоритма, т.е. не будет выполняться основное свойство алгоритма — результативность.

Примером такого рода алгоритмов, могут служить алгоритмы и методы приближенного вычисления функций и решения различного рода уравнений

Выполнение такого алгоритма сводится к генерации некоторой числовой последовательности результатов:

$\{x_0, x_1, \ldots, x_k, x_{k+1}, \ldots\}$, где k – это номер итерации, x_k - это значение, полученное на k-ом шаге процесса.

Итерационной формулой в общем виде называется выражение

$$x_{k+1} = f(x_0, x_1, \ldots, x_k),$$

позволяющее генерировать последующие члены последовательности через вычисленные ранее (на предыдущих шагах алгоритма). Чаще же всего итерационные формулы имеют более простой вид:

$$x_{k+1} = f(x_k)$$

Итерационная последовательность своим пределом должна иметь искомое значение – x*.

$$\lim_{k \to \infty} x_k = x^*$$

Если такой предел существует, то итерационный процесс называется сходящимся. Если нет, то расходящимся.

Реальный вычислительный процесс всегда должен заканчиваться при конечном значении k, поэтому всегда возникает проблема выбора условия окончания итераций – так называемого критерия сходимости.

Вот некоторые общие примеры, используемые на практике:

1. абсолютное изменение параметра на соседних шагах итерационного процесса:

$$|x_k - x_{k-1}| \le \varepsilon$$

2. относительное изменение на соседних шагах

$$\left|\frac{x_k - x_{k-1}}{x_k}\right| \le \varepsilon$$

Здесь ε - наперед заданное малое значение, определяющая точность нахождения решения.

В реализации конкретных численных методов возможно применение специфических критериев или комбинации нескольких критериев.

Таким образом, общая блок-схема итерационных алгоритмов может быть записана так, как представлено на рисунке 1.

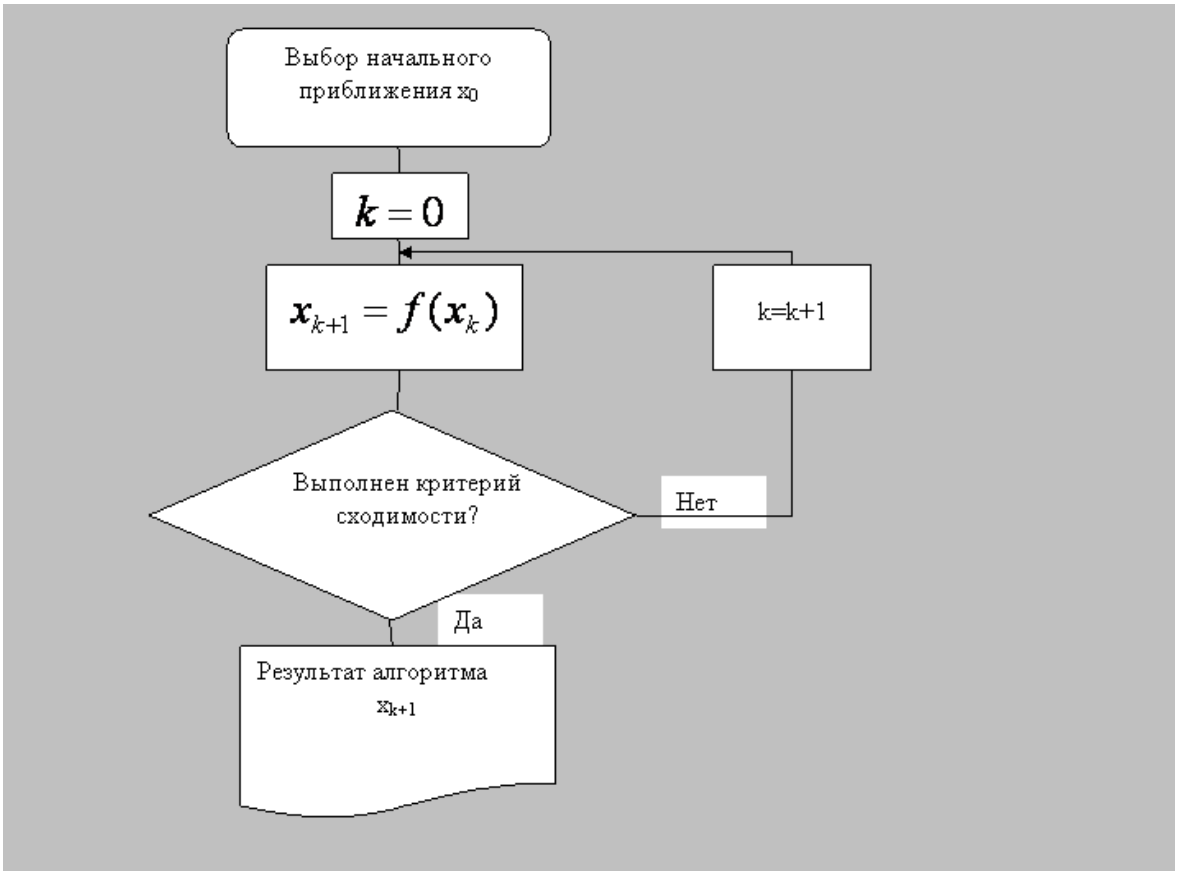

Рисунок 1 – Блок-схема итерационного алгоритма

Приближенные вычисления

Пока что итерационные циклы давали нам «экономию вычислений», избавляя от повторения одних и тех же действий. В вычислительной математике работает другая идея. В методе последовательных приближений координаты очередной точки в пространстве поиска вычисляются через координаты предыдущих точек, а сама последовательность бесконечно приближается (сходится) к некоторой точке с заданным свойством. Число шагов цикла определяется точностью приближения к предельной точке, а оно, в свою очередь, оценивается расстоянием между текущей и предыдущей.

Сама идея, на каких принципах организовать движение от точки к точке, к программированию, как таковому, не относится. Она лежит в предметной области, откуда и ведутся все доказательства работоспособности метода. Для практического программирования важно следующее:

- число шагов цикла (скорость сходимости) определяется самим методом (функцией, вычисляющей следующую точку), а также начальной точкой;
- возможны случаи, когда процесс, наоборот, расходится, т.е. «уходит» от исходной точки, увеличивая значения координат до бесконечности (в программе – переполнение).

Рассмотрим простейший способ нахождения корня функции методом последовательных приближений Для того, чтобы найти корень функции f(x)=0, решается эквивалентное уравнение x = f(x) + x. Если в него подставить точное значение корня, то правая и левая части совпадут. Если же поставить в правую часть любое «постороннее» значение x1, то в левой части получим значение, отличающееся от x1, например x: x=f(x1)+x1.

Если значение x в правой части считать результатом итерационного цикла на предыдущем шаге (x1), а значение x в левой части - результатом текущего шага цикла, то такой процесс последовательного приближения к результату является итерационным.

```
x = x0;
do      {
        x1 = x;
        x = f(x1) + x1;
        } while( условие(x,x1) );
```

То, что это будет последовательность значений x (точек на числовой оси) – это ясно. Если эта последовательность будет сходиться, то она будет сходиться к корню функции – также очевидно. Остается открытым один вопрос, а будет ли она сходиться вообще?

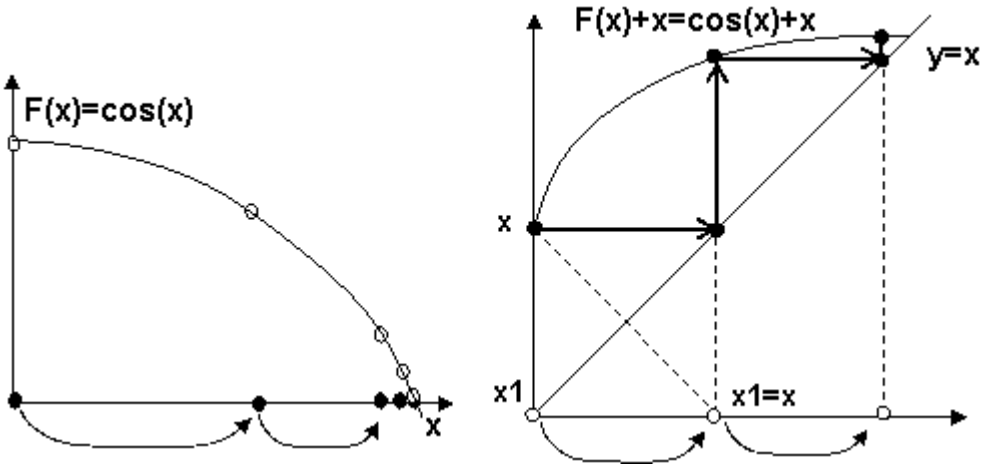

Рисунок 2 - Итерационный цикл приближенного решения уравнения

Графическая иллюстрация, хотя и не является доказательством, но все-таки объясняет, как это работает. Корнем уравнения является пересечение графика функции F(x)+x с диагональю (функция y=x). Когда мы вычисляем очередное значение x=F(x1), мы берем точку на графике функции, соответствующую x1. При присваивании x1=x точка графика проецируется с оси y на ось x, зеркально отражаясь относительно диагонали. Затем от нее берется значение функции на графике. На рисунке мы видим, что такой переход от одной точки на графике к следующей точке можно сделать проще: проводится горизонтальная линия от текущей точки до диагонали, а затем – вертикальная до пересечения с графиком функции. Такая последовательность движения по горизонтами – до пересечения с диагональю и по вертикали – до пересечения с графиком и приводит нас к корню.

Окончательный вид программы включает еще и проверку качественного характера (сходимости) процесса. Дело в том, что данный метод успешно работает не для всех видов функций f() и начальных значений x0. В принципе итерационный процесс может приводить, наоборот, к бесконечному удалению от значения корня. Тогда говорят, что процесс расходится. Для проверки сходимости приходится запоминать разность значений x и x1 предыдущего шага.

```
//-----------------------------------------------------43-03.cpp
//---- Корень функции по методу последовательных приближений
// начальное значение, точность и указатель на внешнюю функцию (см. 9.3)
double find(double x, double eps , double (*pf)(double)){
double x1,dd;
 int n=0;                                          // Счетчик шагов
 dd = 100.;                                  // Предыдущий интервал
           do {
           x1 = x; n++;
           x = (*pf )(x1) + x1;
           printf("n=%d x=%lf y=%lf dx=%lf\n",n,x1,(*pf)(x1),fabs(x1-x));
           if (fabs(x1-x) > dd ) return 0.;     // Процесс расходится
           dd = fabs(x1-x);
           }
 while (dd > eps);
 return x; }                                  // Выход - значение корня

 void main(){ printf("cos(x)=0 x=%lf\n",find(0,0.01,cos)); }
```

Вычисление степенных рядов

Сначала рассмотрим простой вариант с «экономией вычислений». При вычислении значения полинома вида f(x,n) = a0 + a1x + a2x2 + a3x3 +…+ anxn можно избежать многократного вычисления степеней, если представить полином в виде:

$$f(x,n) = ((\ldots((a_n x + a_{n-1})x + a_{n-2})x + \ldots)x + a_1)x + a_0$$

Обозначив накопленную частичную сумму, заключенную в скобках, через s, получим рекуррентную формулу s=s*x+ai, i=n…0.

```
//-----------------------------------------------------43-04.cpp
//-----Степенной полином
 double poly(double A[], double x, int n){
 int i; double s;
 for (s=A[n], i=n; i>=0; i--) s = s * x + A[i];
 return s; }

void main(){ double B[]={4,0,3,1,8}; printf("%lf\n",poly(B,2.,4));}
```

Многие стандартные математические функции вычисляются путем их разложения в степенные ряды (ряды Тейлора []). Точное значение функции определяется как сумма бесконечного ряда f(x)=S0+S1+S2+…Sn+…, слагаемые которого, как правило, содержат степени и факториалы от текущего порядкового номера элемента. Иногда он может содержать другие регулярные выражения (например, произведение четных чисел 2*4*6*…2n, нечетный коэффициент вида 2n-1 и т.п.).

Например, функцию sin(x) можно разложить в такой ряд:

sin(x) = x-x3/3!+x5/5!-x7/7!+…+(-1)n+1x2n+1/(2n+1)!+…

Выражение для n-го слагаемого имеет вид Sn=(-1)n+1x2n-1/(2n+1)!, ряд начинается с n=0, S0=x.

Одним из свойств «хороших» степенных рядов является то, что любое слагаемое превышает оставшуюся бесконечную сумму «хвоста» ряда. На этом основании точность вычисления может быть оценена значением очередного слагаемого.

Для «экономии вычислений» степеней, факториалов и других регулярных выражений выводится рекуррентная формула, позволяющая вычислить текущее слагаемое через предыдущее. Это можно сделать аналитически, разделив в общем виде выражение Sn на Sn-1

$$K(x,n)=Sn/Sn-1=((-1)^{n+1}x^{2n+1}/(2n+1)!)/((-1)^n x^{2n-1}/(2n-1)!)$$
$$K(x,n)=-x^2/(2n(2n+1))$$

Формулу можно вывести индуктивно, рассматривая подряд несколько пар соседних слагаемых. Для каждой пары выводится коэффициент перехода, который потом приводится к общему виду:

n	0	1	2	3		n
Sn	x	$-x^3/3!$	$x^5/5!$	$-x^7/7!$		$-x^{2n-1}/(2n-1)!$
k(x)	Домножить предыдущий	$-x^2/(2*3)$	$-x^2/(4*5)$	$-x^2/(6*7)$		$-x^2/(2n*(2n+1))$

В таблице записывается, на что нужно умножить предыдущее слагаемое, чтобы получить текущее. Обратите внимание, что значение n берется для текущего столбца, например, 3 и 7 находятся в таком же соотношении, что и n с 2n+1.

```
//-----------------------------------------------------43-05.cpp
//--- Вычисление значения функции sin через степенной ряд
 double sum(double x,double eps,int &n){
 double s,sn;                        // Сумма и текущее слагаемое ряда
     for (s=0., sn = x, n=1; fabs(sn) > eps; n++) {
     s += sn;
     sn = - sn * x * x / (2.*n * (2.*n + 1));
     }
 return s;}
 // Вычисление степенного ряда для x в диапазоне от 0.1 до 1 с шагом 0.1
 void main(){
 double x,y;
 int nn;
 for (x=0.1; x <= 1.; x += 0.1){
          y=sum(x,0.0001,nn);
          printf("n=%d x=%0.1lf\t sum=%0.4lf\t sin=%0.4lf\n",nn,x,y,sin(x));
          }}
```

В программе текущее слагаемое sn каждый раз умножается на переходный коэффициент для получения следующего значения. Перед этим текущее значение добавляется к сумме s. В цикле легко ошибиться «на один шаг», поэтому необходимо обратить внимание на начало его выполнения: значение sn=x устанавливается для слагаемого при n=0, а на первом шаге вычисляется значение sn уже для n=1.

Примеры использования итерационного алгоритма

В итерационных алгоритмах решение задачи реализуется путем последовательного приближения к искомому результату[1]. Процесс является циклическим, поскольку заключается в многократных вычислениях. Начальное приближение Y0 выбирается заранее или задается по определенным правилам. Заканчивается итерационное вычисление при выполнении условия

$|Yi - Yi-1| < d$

где d - допустимая ошибка вычисления.

Типовая структура алгоритма итерационных вычислений имеет вид, показанный на рисунке 3.

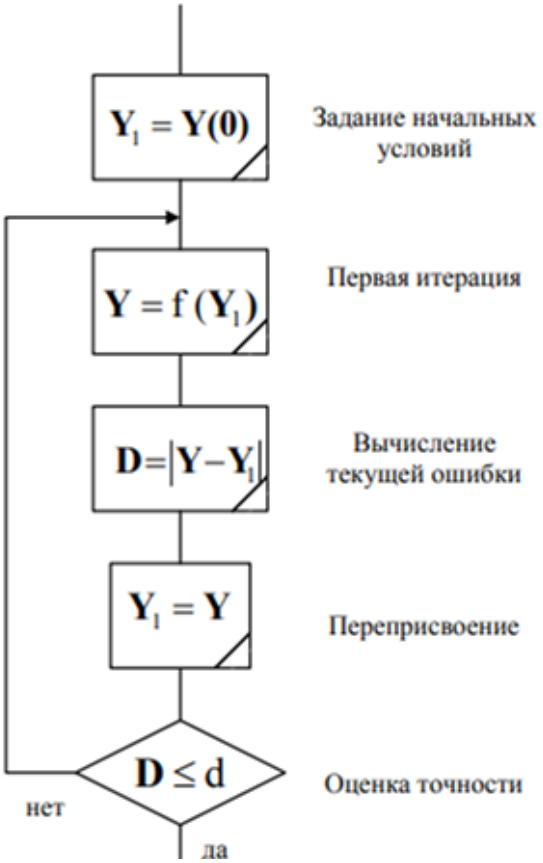

Рисунок 3 – Структура итерационного алгоритма

Пример итерационного алгоритма

Составить алгоритм вычисления функции y= √x с точностью d, используя рекуррентную формулу

$y_{i+1}=0.5*(x/y_i+y_i)$.

Этот метод называется методом Ньютона, но начало получил из Древней Греции и приписывается Герону Александрийскому.

Заслуга метода Ньютона в том, что он описал данный механизм нахождения квадратного корня с использованием формул, которые в дальнейшем стали называться как итерационными формулами.

Найдем приближенное значение квадратного корня из 720.

Ближайшее к 720 число, из которого извлекается квадратный корень, есть число 729, оно имеет корнем 27. Разделив 720 на 27, получаем $26\frac{2}{3}$. Найдем среднее арифметическое чисел 27 и $26\frac{2}{3}$.

Получаем $(26\frac{2}{3} + 27) : 2 = 53\frac{2}{3} : 2 = 26\frac{5}{6}$

Это и есть результат. Если возвести это число в квадрат, получим $720\frac{1}{36}$.

Где $\frac{1}{36}$ -это погрешность алгоритма.

Если начальное приближение y1 = x , тогда на первом цикле вычисления будем иметь

y 1=0.5*(x/y1+y1)

Блок - схема алгоритма решения примера 4 приведена на рисунке 2.

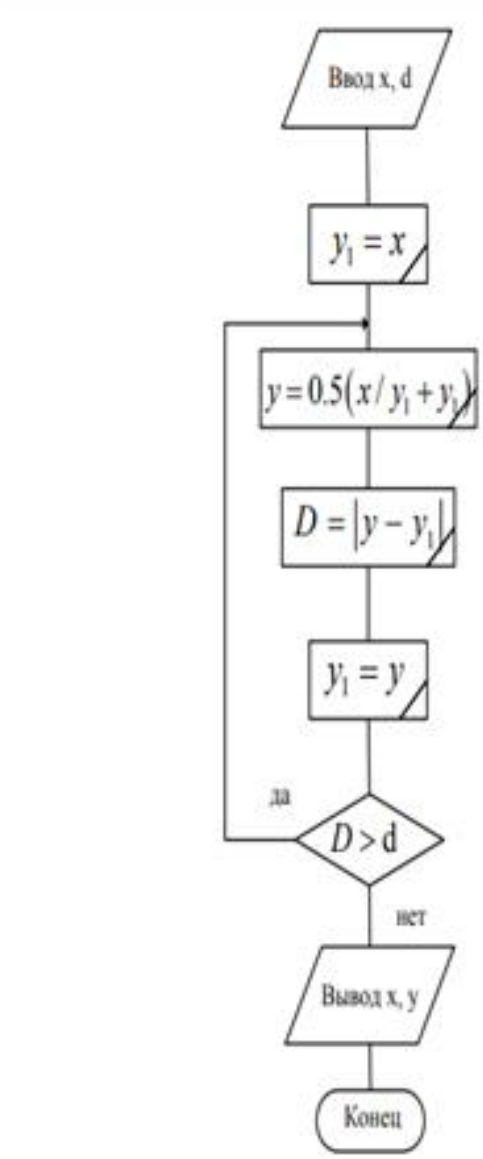

Рисунок 2-Блок-схема алгоритма

Приведем пример вычисления корня уравнения методом дихотомии или половинного деления.

Перед применением метода для поиска корней функции необходимо отделить корни одним из известных способов, например, графическим методом. Отделение корней необходимо в случае, если неизвестно на каком отрезке нужно искать корень.

Будем считать, что корень t функции $f(x)=0$ отделён на отрезке $[a,b]$. Задача заключается в том, чтобы найти и уточнить этот корень методом половинного деления. Другими словами, требуется найти приближённое значение корня с заданной точностью ϵ.

Пусть функция f непрерывна на отрезке $[a,b]$,

$f(a)\cdot f(b)<0,\ \epsilon=0{,}01$ и $t\in[a,b]$ - единственный корень уравнения $f(x)=0,\ a\le t\le b$.

(Мы не рассматриваем случай, когда корней на отрезке $[a,b]$ несколько, то есть более одного. В качестве ϵ можно взять и другое достаточно малое положительное число, например, $0{,}001$.)

Поделим отрезок $[a,b]$ пополам. Получим точку $c=\frac{a+b}{2},\ a<c<b$ и два отрезка $[a,c],\ [c,b]$.

Если $f(c)=0$, то корень t найден ($t=c$).

Если нет, то из двух полученных отрезков $[a,c]$ и $[c,b]$ надо выбрать один $[a_1;b_1]$ такой, что $f(a_1)\cdot f(b_1)<0$, то есть

$[a_1;b_1]=[a,c]$, если $f(a)\cdot f(c)<0$ или

$[a_1;b_1]=[c,b]$, если $f(c)\cdot f(b)<0$.

Новый отрезок $[a_1;b_1]$ делим пополам. Получаем середину этого отрезка $c_1=\frac{a_1+b_1}{2}$ и так далее.

Для того, чтобы найти приближённое значение корня с точностью до $\epsilon>0$, необходимо остановить процесс половинного деления на таком шаге n, на котором $|b_n-c_n|<\epsilon$ и вычислить $x=\frac{a_n+b_n}{2}$. Тогда можно взять $t\approx x$.

Начало

e:=0.001

c:=(a+b)/2

F(a)*F(c)<=0

Да

b:=c

a:=c

abs(b-a)<e ;

Да

x:=(a+b)/2

x

Конец