Тема 3
Программирование разветвленных алгоритмов

Ветвления - алгоритмы с несколькими сценариями развития событий

- если капает вода из крана, то кран нужно закрыть;
- если на счете достаточно денег, можно провести операцию оплаты; если денег не хватает, оплата не может быть проведена и надо вывести ошибку пользователю.

Особенности типа bool

Формально в bool используются только два литерала: true и false (истина и ложь).

Фактически, в переменную типа bool можно записать любое число, которое будет неявно приведено к логическому типу по принципу:

1) целое 0 интерпретируется как false (в т.ч. +0/-0);

2) любое число, не совпадающее с нулем интерпретируется как true.

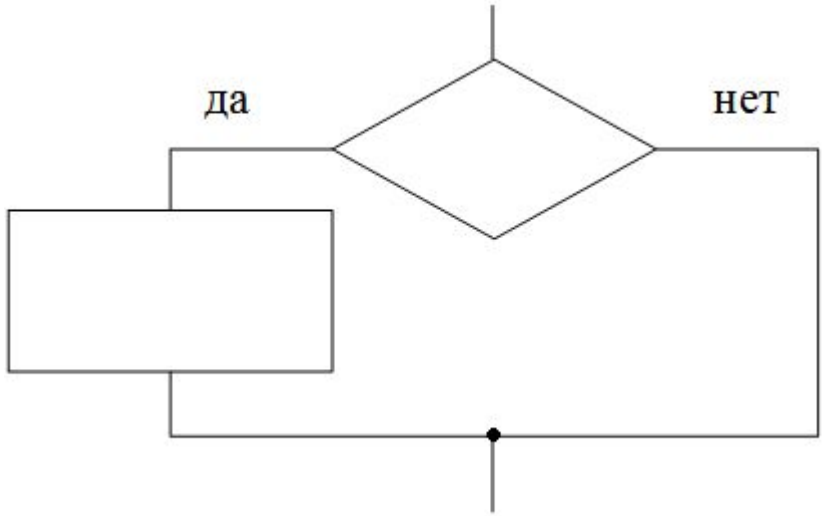
Стандартный ввод работает таким же образом.

Стандартный вывод выводит значения 0 или 1. Для вывода на экран значений true / false можно воспользоваться манипулятором boolalpha (C++11)

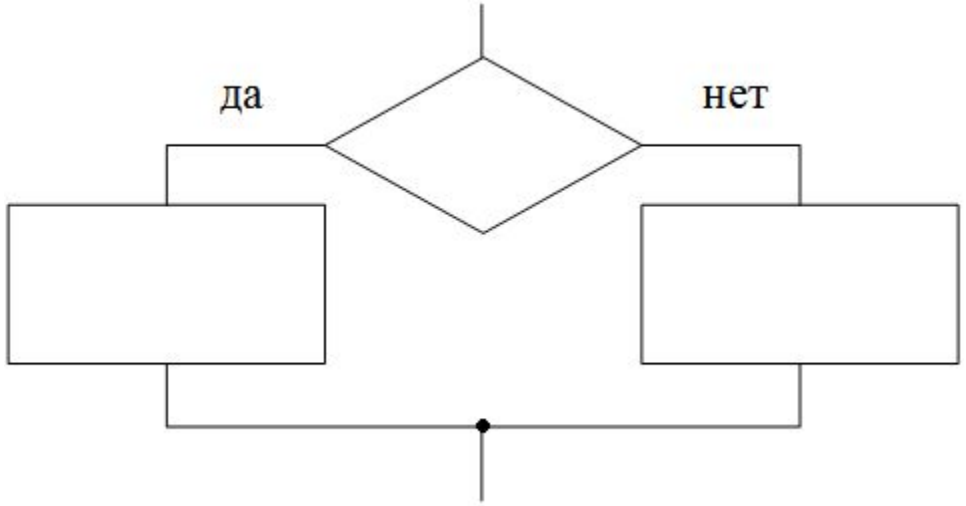
```
int negZero = -0;
float floatNearZero = -0.00000001;
cout << bool(negZero) << endl;
cout << boolalpha << bool(floatNearZero);
```

Неполный и полный условные операторы

неполный



полный



Синтаксис условного оператора

Условие оператора if должно иметь тип bool. Для прочих типов в условии будет выполнено неявное преобразование к bool.

```
if (условие)
   оператор1; //неполный

if (условие)
   оператор1;
else
   оператор2; //полный
```

Пример условного оператора

```
if (a < b)
   cout << "a меньше";    //неполный

if (a < b)
   cout << "a меньше";
else                  //полный
   cout << "b меньше или равно";
```

Операторы отношения на примере стандартных типов

```
float floatA = 1.1, floatB = 1.1;
char charA = 'r', charB = 'z';
string stringA = "i am excited about",
       stringB = "programming";
bool boolFloat = floatA == floatB;    // true
bool boolChar = charA > charB;        // false
bool boolString = stringA < stringB;  // true
```

*Операторы отношения имеют разные ранги приоритета:

<, <=, >, >= - ранг 9;

==, != - ранг 10.

Пример нахождения меньшего из двух чисел

```
int main(){
float a, b;
cin >> a >> b;
float min;
if(a < b)
   min = a;
else
   min = b;
cout << min;
}
```

Условия без использования операторов отношения

```
if (x){ // равносильно x != 0
/*действие, если x не равно 0*/
}

if (!x){ // равносильно x == 0
/*действие, если x равно 0*/
}
```

Пример проверки деления на 0

```
int main(){
float a, b;
cin >> a >> b;
if(b)
   cout << a / b;
else
   cout << "b не должно быть равно нулю!";

/*то же что
if (!b)
   cout << "b не должно быть равно нулю!";
else
   cout << a / b;
*/
}
```

Текущий контроль

Что будет выведено на экран при выполнении следующего кода?

```
if (4 > 3)
    cout << 4;
if (2 > 3)
    cout << 2;
```

1) 1
2) 2
3) 3
4) 4

Текущий контроль

Что будет выведено на экран при выполнении следующего кода?

```
if (4 > 3)
    cout << 4;
if (2 > 3)
    cout << 2;
```

1) 1
2) 2
3) 3
4) **4**

Текущий контроль

Что будет выведено на экран при выполнении следующего кода?

```
string h = "hello";
string hw = "hello, world";
if (h == hw)
    cout << h;
if (h != hw)
     cout << "test"
```

1) hello test
2) hello
3) test
4) h
5) будет синтаксическая ошибка

Текущий контроль

Что будет выведено на экран при выполнении следующего кода?

```
string h = "hello";
string hw = "hello, world";
if (h == hw)
    cout << h;
if (h != hw)
     cout << "test"
```

1) hello test
2) hello
3) test
4) h
5) **будет синтаксическая ошибка**

Текущий контроль

Что будет выведено на экран при выполнении следующего кода?

```
cout << (2 >= 2);
cout << (15 <= 16);
```

1) 00
2) 11
3) 10
4) 01
5) ничего, будет синтаксическая ошибка

Текущий контроль

Что будет выведено на экран при выполнении следующего кода?

```
cout << (2 >= 2);
cout << (15 <= 16);
```

1) 00
2) **11**
3) 10
4) 01
5) ничего, будет синтаксическая ошибка

Логические операции

Логическое «и» && and (ранг приоритета 14)

Логическое «или» || or (ранг приоритета 15)

Логическое «не» ! not (ранг приоритета 3)

Ленивое* поведение для операторов && ||

Логическое и		
0	0	0
0	1	0
1	0	0
1	1	1

Логическое или		
0	0	0
0	1	1
1	0	1
1	1	1

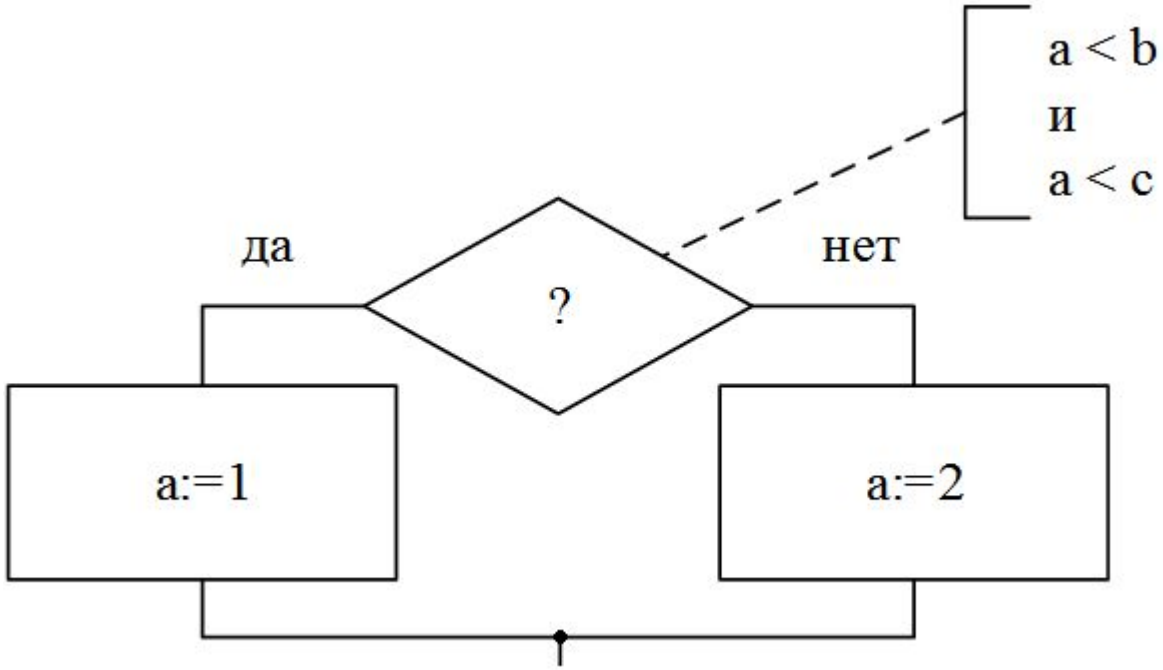
Логическое не	
0	1
1	0

* оператор && проверяет второй аргумент, только если 1-й истина

оператор || не проверяет второй аргумент, только если 1-й истина

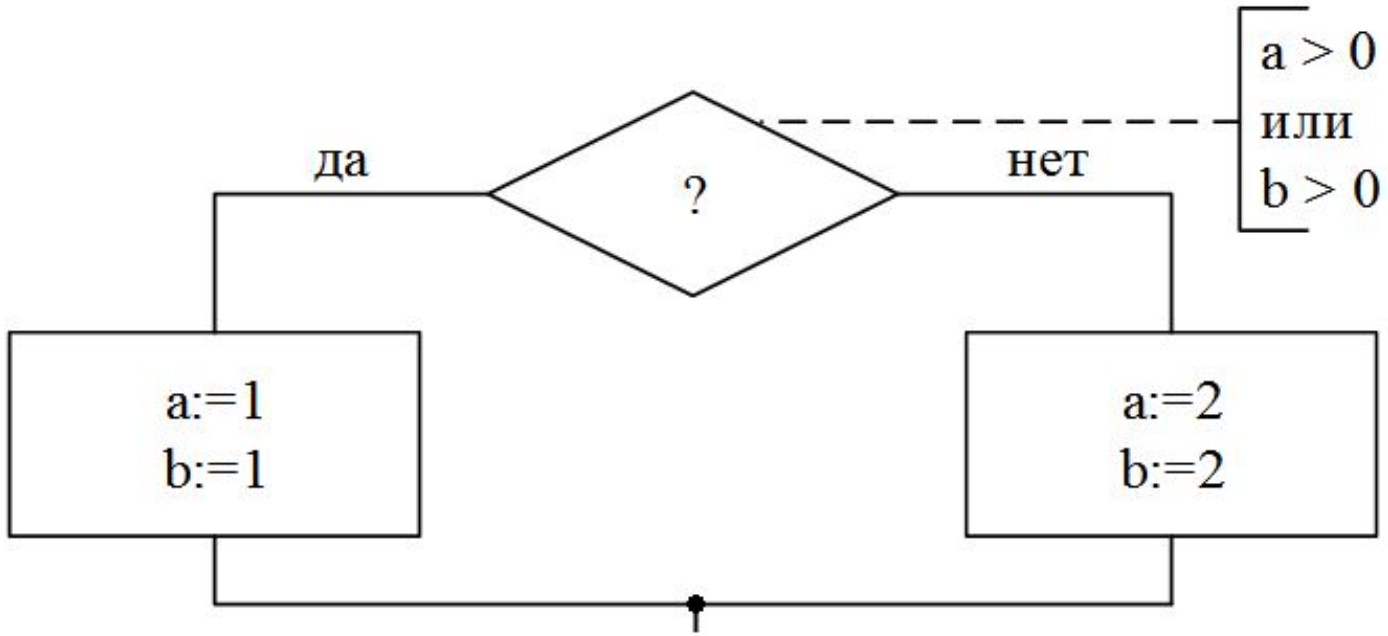
Пример логического «и»

```
if (a < b && a < c)
    a = 1;
else
    a = 2;


if (0 < x < 10){}


if (0 < x && x < 10){}
```



Пример логического «или»

```
if (a > 0 || b > 0)
    a = b = 1;
else
    a = b = 2;
```



Применение логических операций

```
bool flag = true;
flag = !flag;
int a, b;
cin >> a >> b;
bool bothNegative = a < 0 && b < 0;
bool atOnceOneNegative = a < 0 || b < 0;
```

Текущий контроль

Что будет выведено на экран при выполнении следующего кода?

```
if ((6 > 4) && (7 == 8))
     cout << "and";
if ((6 > 4) or (7 == 8))
     cout << "or";
```

1) and
2) or
3) andor
4) будет синтаксическая ошибка

Текущий контроль

Что будет выведено на экран при выполнении следующего кода?

```
if ((6 > 4) && (7 == 8))
     cout << "and";
if ((6 > 4) or (7 == 8))
     cout << "or";
```

1) and
2) **or**
3) andor
4) будет синтаксическая ошибка

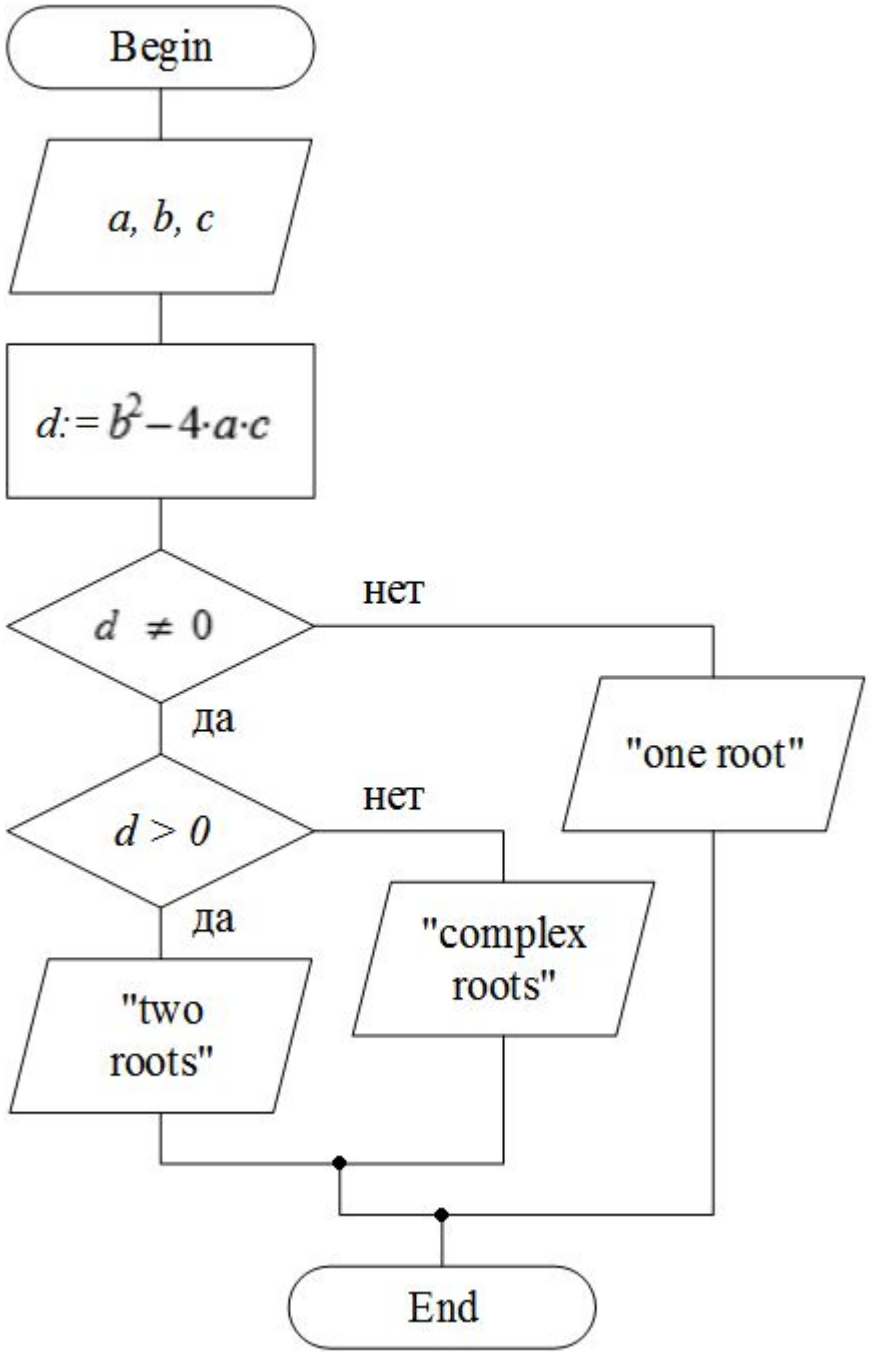
Операция присваивания в проверяемом условии

Синтаксически не запрещается использовать присваивание внутри условия, но надо понимать принцип работы операторов

```
int a, b, c, d;
cin >> a >> b >> c;
if (d = b * b – 4 * a * c)
   cout << "two roots";
else
   cout << "one root";
cout << ", D = " << d << endl;
```

Вложенные условия. Пример 1

```
int a, b, c, d;
cin >> a >> b >> c;
if (d = b * b – 4 * a * c)
   if (d > 0)
      cout << "two roots";
   else
      cout << "complex roots";
else
   cout << "one root";
```



Преимущества использования вложенных условий

- Для некоторых задач использовать вложения условий более оптимально, чем применение нескольких неполных условных операторов.

Задача вычисления значения кусочно-заданной функции f с тремя промежутками значений x

$$F(x) = \begin{cases} x, \text{при } x < 0, \\ 2x, \text{при } 0 \leq x < 1, \\ 3x, \text{при } x \geq 1. \end{cases}$$

Использование отдельных операторов if для проверки принадлежности к каждому из промежутков дает **в худшем случае проверку четырех условий**. Использование вложенных if позволит сократить проверку до **двух условий**.

Преимущества использования вложенных условий

```
//плохое решение – проверка четырех условий
if (x < 0)
   f = x;
if (x >= 0 && x < 1)
   f = 2 * x;
if (x >= 1)
   f = 3 * x;
//хорошее решение – проверка двух условий
if (x < 0)
   f = x;
else
   if (x < 1)
      f = 2 * x;
   else
      f = 3 * x;
```

Составной оператор (фигурные скобки)

Необходим для объединения нескольких действий при выполнении условия.

```
if (int d = b * b – 4 * a * c)
   if (d > 0){
      float x1 = ((-1) * b + sqrt(d)) / 2 / a;
      float x2 = ((-1) * b - sqrt(d)) / 2 / a;
      cout << "two roots: x1 = " << x1 << ", x2 = " << x2;
      }
   else
      cout << "complex roots";
else
   cout << "one double root: x = " << -b / (2 * a);
```

*d - локальная переменная, вне оператора if она не видна

Оператор запятая в условном операторе

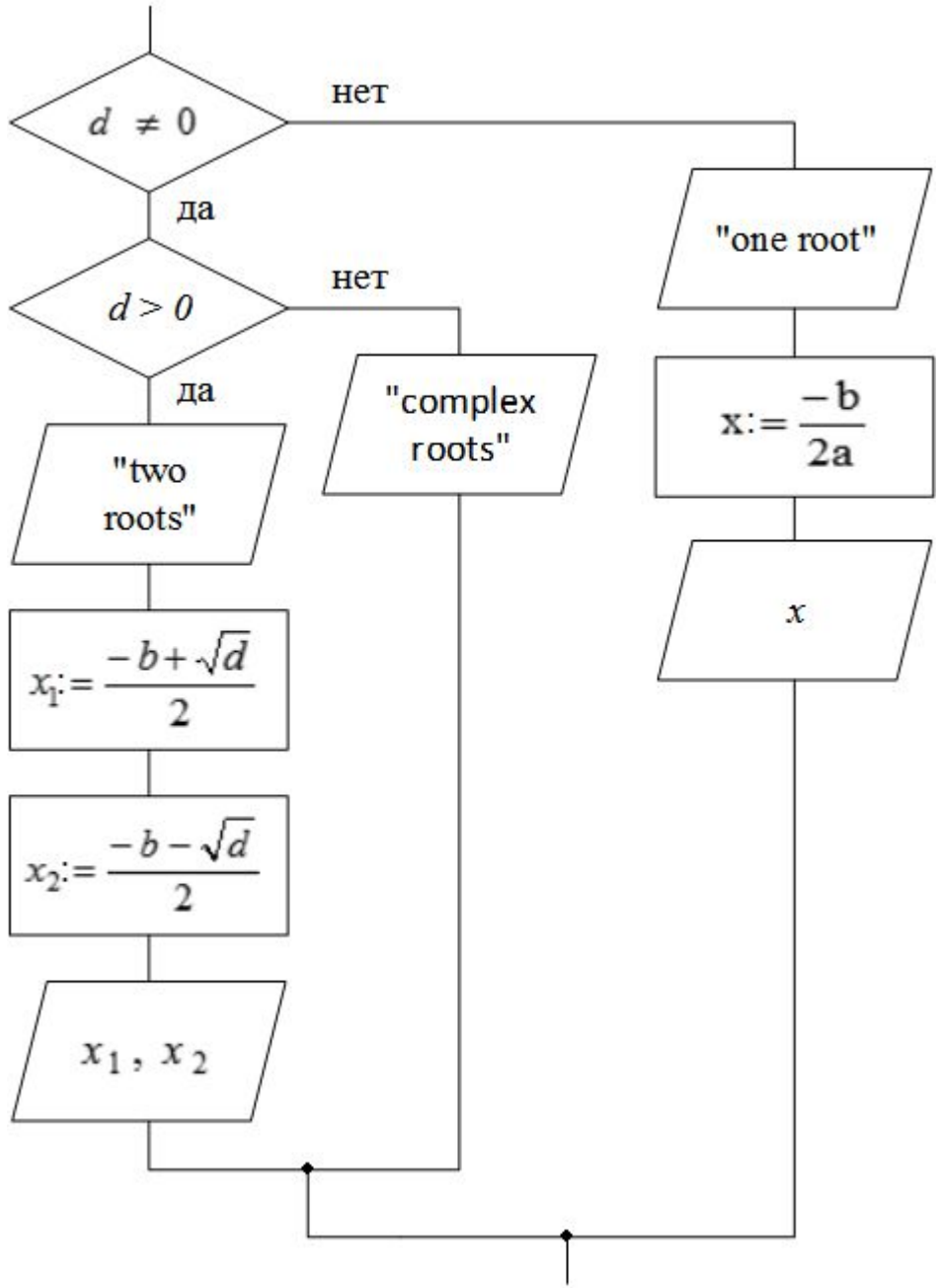
Связывает несколько выражений, означает последовательность действий.

```
float d;
if (d = b * b – 4 * a * c, d > 0){
   float x1 = ((-1) * b + sqrt(d) ) / 2;
   float x2 = ((-1) * b - sqrt(d) ) / 2;
   cout << "two roots: x1 = " << x1 << ", x2 = " << x2;
   }
else
   if(!d)
      cout << "one double root: x = " << -b / 2 / a;
   else
      cout << "complex roots";
```

Фрагмент алгоритма решения задачи

схема вложенности:

```
if
    if
    else
else
```

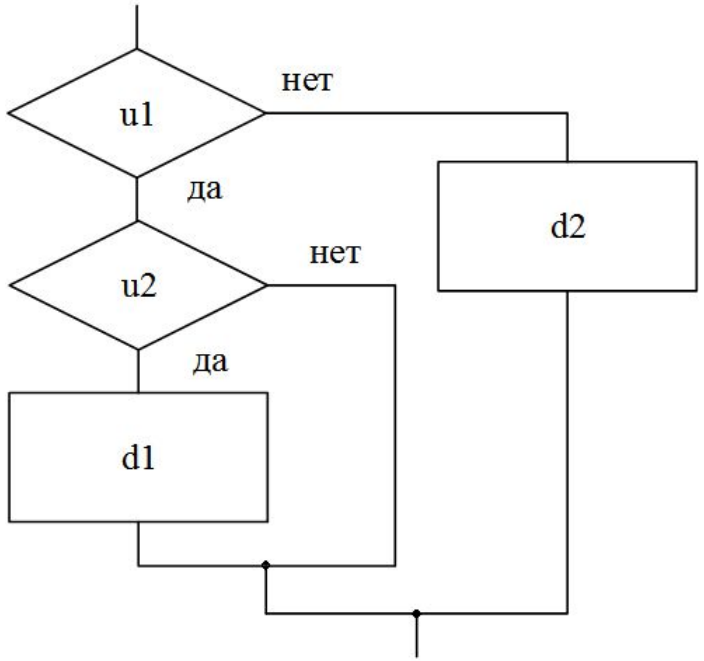


Особенности вложения условий

Компилятор будет пытаться связать else с ближайшим вышестоящем, не занятому другим else оператором if.

В случае, когда внешний полный условный оператор содержит внутренний неполный условный оператор, внутренний неполный условный оператор **надо обязательно** обернуть фигурными скобками, иначе компилятор отнесет else к внутреннему if.

```
if (u1){
  if (u2)
  d1;
  }
else
  d2;
```



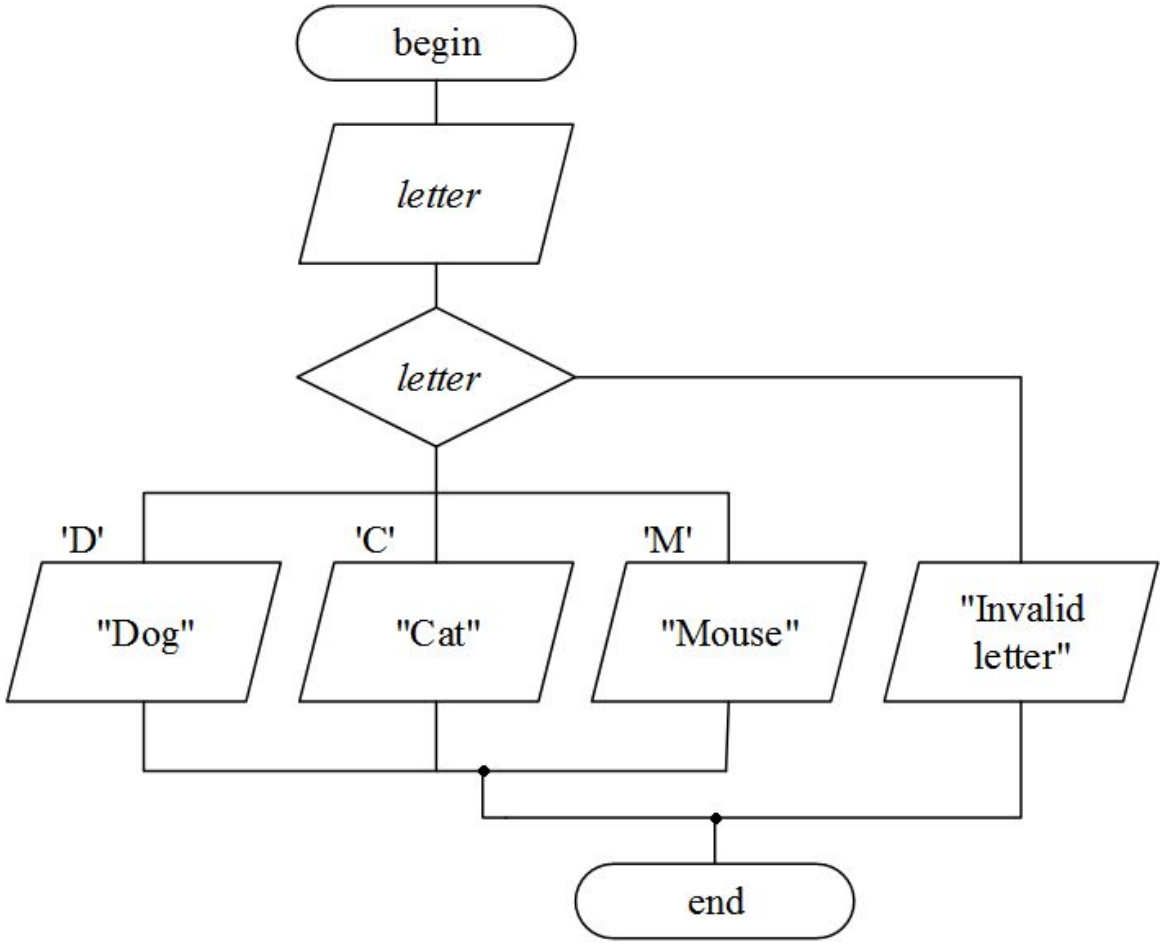
Оператор switch

Применяется, когда необходимо выполнить цепочку проверок if-else на равенство некоторым значениям.
Меток case в теле switch может быть сколько угодно. Если значение ни одной из меток не совпало со значением в выражении-селекторе, то выполняется группа операторов, стоящая за меткой default (default не является обязательным элементом оператора).

```
switch (выражение){
case константное_выражение1:
   группа_операторов1;
   break;
case константное_выражение2:
   группа_операторов2;
   break;
default:
   группа_операторов;
   break;
}
```

Фрагмент кода, содержащий оператор выбора

```
char letter;
cin >> letter;
switch (letter){
   case 'D' :
      cout << "Dog"; break;
   case 'C' :
      cout << "Cat"; break;
   case 'M' :
      cout << "Mouse"; break;
   default  :
      cout << "Invalid letter";
   }
```

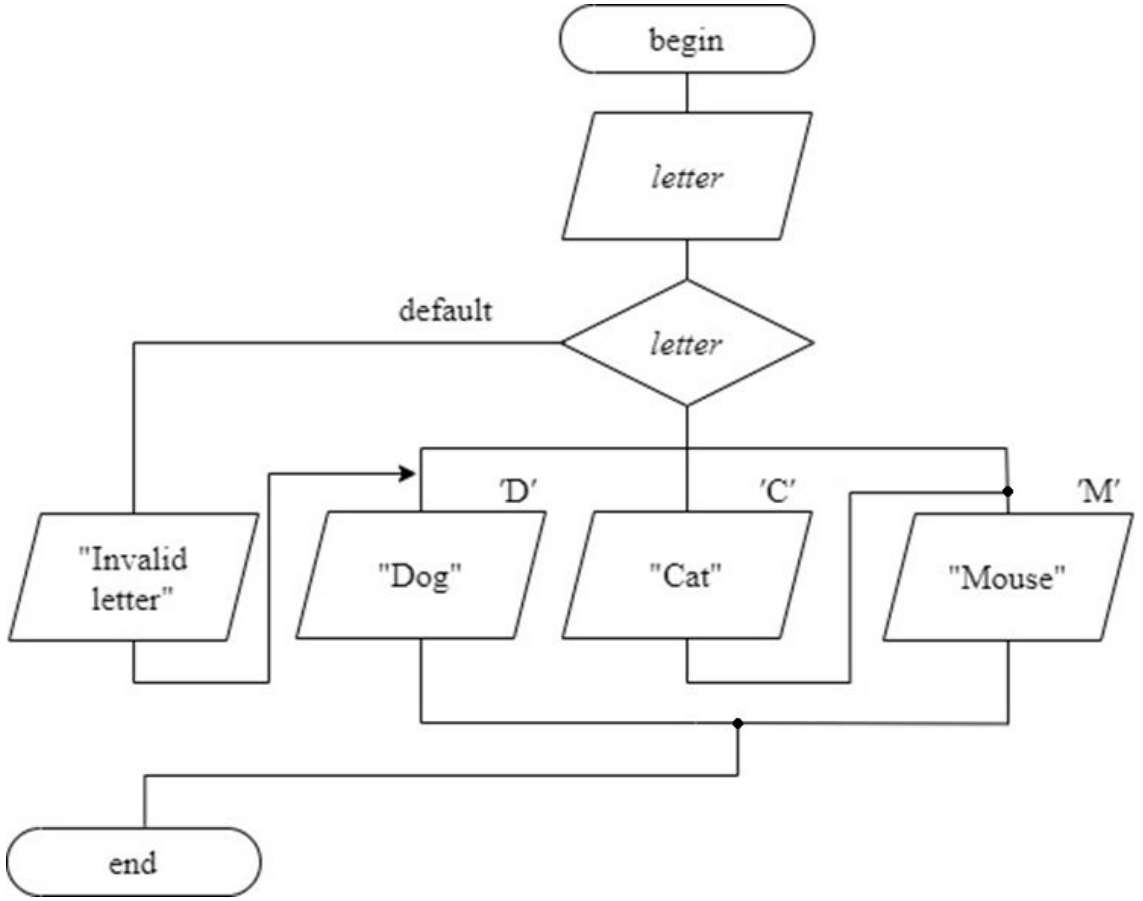


Несколько меток case в одной инструкции switch

```
char letter;
cin >> letter;
switch (letter){
   default  :
      cout << "Invalid letter"; break;
   case 'D': case 'd':
      cout << "Dog"; break;
   case 'C': case 'c':
      cout << "Cat"; break;
   case 'M': case 'm':
      cout << "Mouse"; break;
   }
```

Пример switch без break для некоторых case

```
char letter;
cin >> letter;
switch (letter){
   default:
      cout << "Invalid letter";
   case 'D': case 'd':
      cout << "Dog"; break;
   case 'C': case 'c':
      cout << "Cat";
   case 'M': case 'm':
      cout << "Mouse"; break;
   }
```



Пример использования целого типа в switch

```
int digit;
cin >> digit;
switch (digit){
   case 1 : cout << "One"; break;
   case 2 : cout << "Two"; break;
   case 3 : cout << "Three"; break;
   case 4 : cout << "Four"; break;
   case 5 : cout << "Five"; break;
   case 6 : cout << "Six"; break;
   case 7 : cout << "Seven"; break;
   case 8 : cout << "Eight"; break;
   case 9 : cout << "Nine"; break;
   case 0 : cout << "Zero"; break;
   default  : cout << "Invalid digit";
 }
```

Пример использования перечислений в switch

```
enum Color{red, orange, yellow, green, skyBlue, blue,
      margenta};

int main(){
Color x;
cin >> x; //integer value range 0..6
switch (x){
   case red: case orange: case yellow:
      cout << "warm color spectrum"; break;
   case green: case skyBlue: case blue: case margenta:
      cout << "cold color spectrum"; break;
   default:
      cout << "no such color";
   }
}
```

Правила описания оператора switch

- значение варианта выбора должно иметь тип bool, int, char или enum;
- значения меток case должны соответствовать по типу значению варианта выбора;
- значения меток case должны быть константными выражениями;
- метки двух разделов case не должны иметь одинаковые значения;
- один раздел case может иметь несколько меток;
- каждый раздел case желательно завершать ключевым словом break.

Если после какой-либо группы операторов case не указан break, то «switch проваливается в следующий кейс», это значит что группы операторов для всех следующих кейсов будут выполняться без проверки соответствия значения метки селектору до тех пор пока не будет обнаружен break, либо конец switch.

Текущий контроль

Что будет выведено на экран при выполнении следующего кода?

```
char letter = 'b';
switch (letter){
   case 'D': case 'd':
      cout << "Dan";
   case 'B': case 'b':
      cout << "Bob";
   default :
      cout << "Invalid letter";
   case 'M': case 'm':
      cout << "Mike"; break;
   }
```

1) Bob
2) Invalid letter
3) BobInvalid letter
4) BobInvalid letterMike
5) BobInvalid letterMikeDan

Текущий контроль

Что будет выведено на экран при выполнении следующего кода?

```
char letter = 'b';
switch (letter){
   case 'D': case 'd':
      cout << "Dan";
   case 'B': case 'b':
      cout << "Bob";
   default :
      cout << "Invalid letter";
   case 'M': case 'm':
      cout << "Mike"; break;
   }
```

1) Bob
2) Invalid letter
3) BobInvalid letter
4) **BobInvalid letterMike**
5) BobInvalid letterMikeDan

Побитовые операции

```
int a = 3, b = 5, c;
```

или	|	(ранг 13)	`c = a | b;  // c = 7`
исключающее или	^	(ранг 12)	`c = a ^ b;  // c = 6`
и	&	(ранг 11)	`c = a & b;  // c = 1`
сдвиг вправо	>>	(ранг 7)	`c = b >> 1;  // c = 2`
сдвиг влево	<<	(ранг 7)	`c = a << 2;  // c = 12`
не	~	(ранг 3)	`c = ~b;  // c = -6`

Модификации оператора присваивания (ранг 16) :

|= &= ^= >>= <<=

Тернарный оператор условия ?:

условие ? оператор1 : оператор2; (ранг приоритета 16)

```
// с помощью оператора if
if(a < b)
   min = a;
else
   min = b;
// с помощью тернарного оператора условия
a < b ? min = a : min = b;
// или
min = a < b ? a : b;
```

Задача. Проверка логина и пароля

Даны два имени пользователя (user1, user2) и пароли этих пользователей соответственно (password1 и password2).

Необходимо написать программу, которая принимает от пользователя логин и пароль, проверяет их и выводит одно из следующих сообщений:

- "successful connection", если проверка прошла успешно;

- "unsuccessful connection", если проверка не прошла;

- "unknown user", если введено неизвестное программе имя пользователя.

Решение задачи Проверка логина и пароля

```
string user1Log = "user1", user2Log = "user2";
string user1Pass = "password1", user2Pass = "password2";
string login, password;
cin >> login;
if (not (login == user1Log or login == user2Log))
   cout << "unknown user";
else{
   cin >> password;
   bool user1 = login == user1Log and password == user1Pass;
   bool user2 = login == user2Log and password == user2Pass;
   cout << (user1 or user2 ? "successful connection"
                : "unsuccessful connection");
   }
```

Примеры решения задач

1. Программирование разветвленных алгоритмов с помощью неполного условного оператора if

Дано целое число *x*. Проверить является ли оно четным

2. Программирование разветвленных алгоритмов с помощью полного условного оператора if.

Даны четыре целых числа *a*, *b*, *c*, *d*. Вывести сообщение «Верно», если при делении числа *a* на число *b* (считать, что $b \neq 0$) получается остаток равный одному из заданных чисел *c* или *d,* и возвести в квадрат числа *a* и *b* в противном случае.

3. Программирование разветвленных алгоритмов с помощью вложенных конструкций оператора if

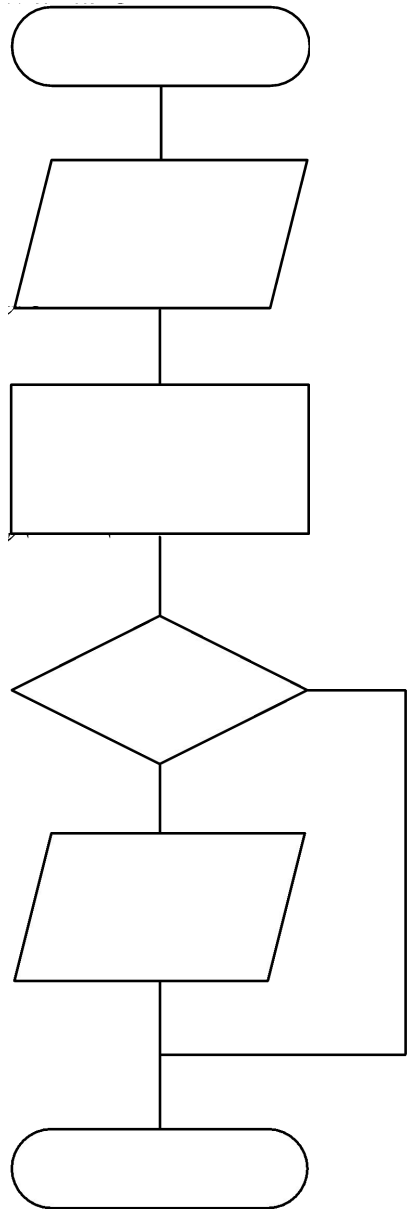
Наименьшее из трех действительных чисел заменить их полусуммой, наибольшее удвоенным произведением.

4. Программирование разветвленных алгоритмов с помощью оператора выбора switch

Даны два действительных числа *a* и *b*. Выполнить над ними соответствующее действие согласно введенному знаку операции: +, -, *, /.

Решение задачи 1

```
int main(){
long x;
cout << "input integer value\n";
cin >> x;
if (x % 2 == 0)
   cout << "является";

// можно использовать запись if (!(x % 2))
// cout << "является";
}
```

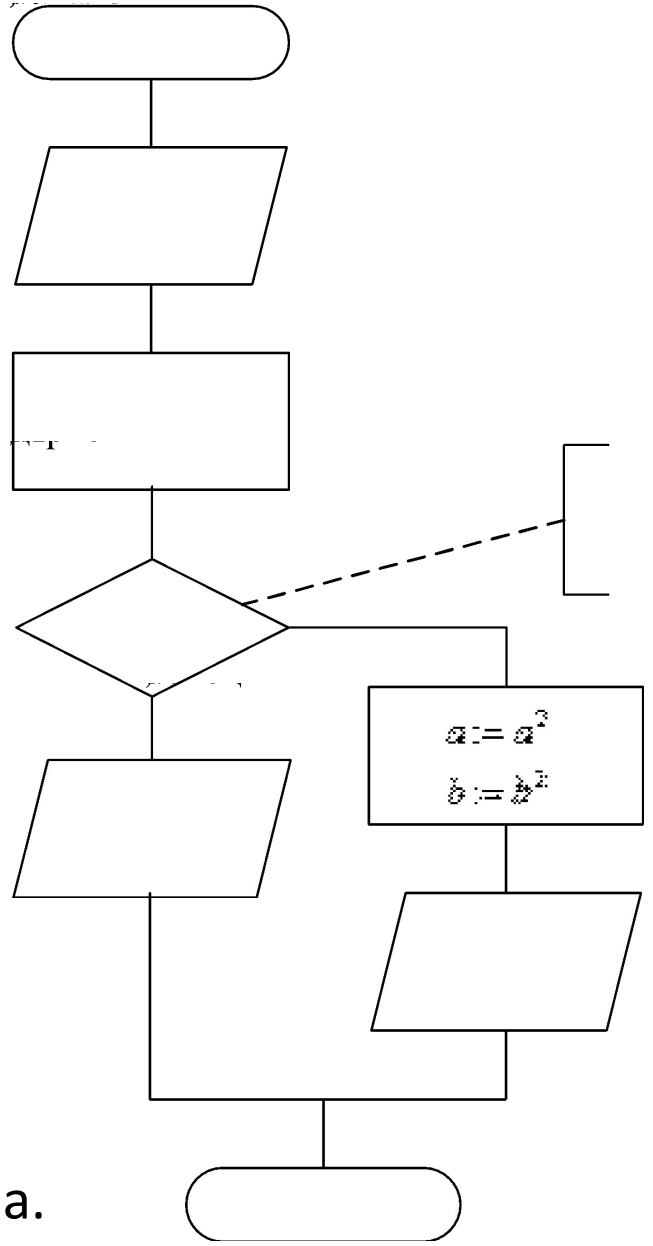


Входные данные: *x* – целое число.

Выходные данные: сообщение *«Является»*.

Решение задачи 2

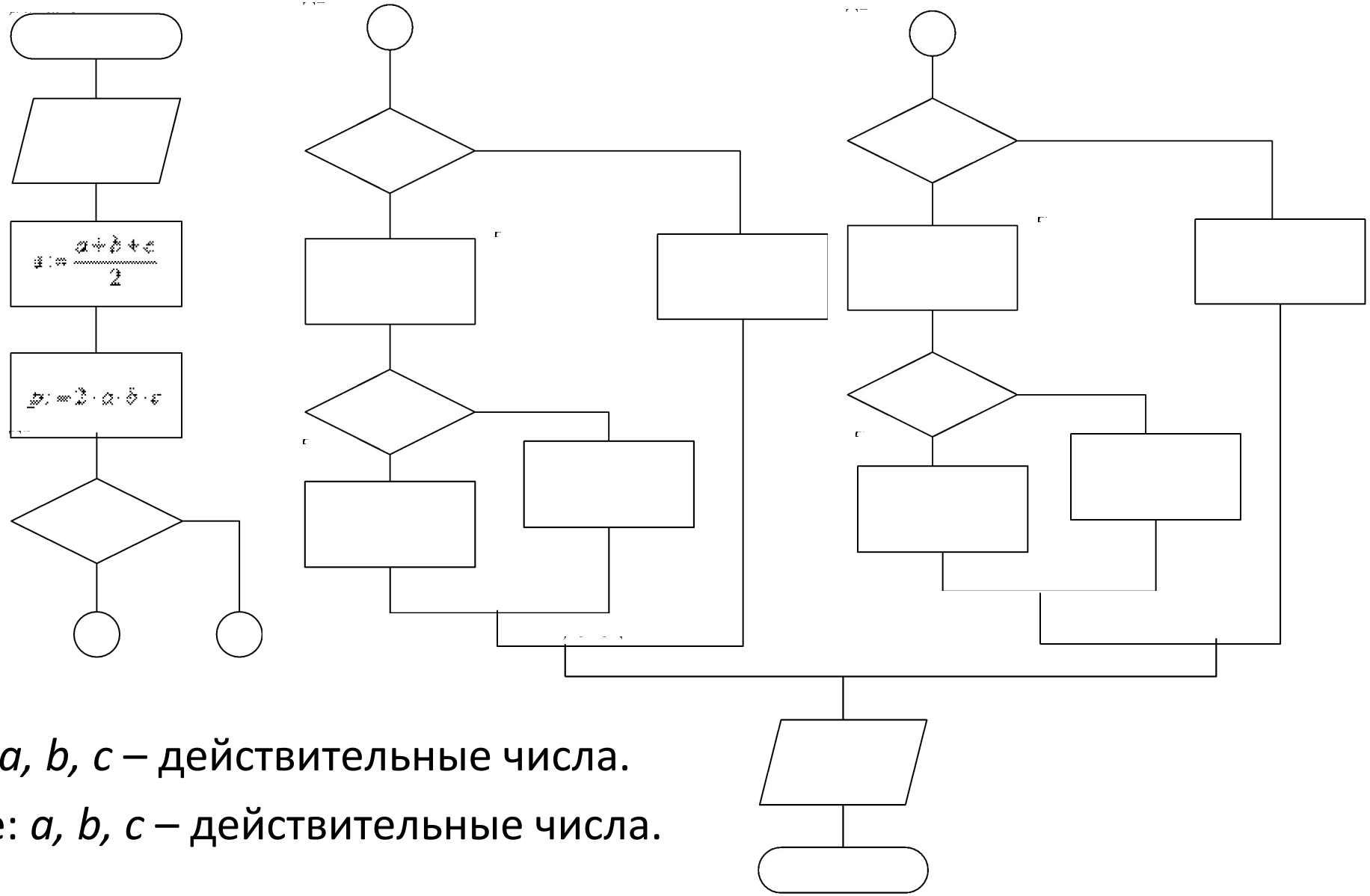
```
int main(){
int a, b, c, d;
cout << "input 4 integer values";
cin >> a >> b >> c >> d;
if ((a % b == c) || (a % b == d))
   cout << "Верно";
else{
   a *= a;
   b *= b;
   cout << "a = " << a << "b = " << b;
   }
}
```



Входные данные: *a, b, c, d* – целые числа.

Выходные данные: сообщение *«верно»* или *a, b* – целые числа.

Решение задачи 3



Входные данные: *a, b, c* – действительные числа.

Выходные данные: *a, b, c* – действительные числа.

```
int main(){
float a, b, c;
cout << "input 3 float values\n";
cin >> a >> b >> c;
float s = (a + b + c) / 2;
float p = 2 * a * b * c;
if (a < b){//abc acb cab
   if (a < c){//abc acb
      a = s;
      if (b < c)
         c = p;
      else
         b = p;
      }
   else{//cab
      c = s;
      b = p;
      }
   }
else {//bac bca cba
   if (b < c){
      b = s;
      if (a < c)
         c = p;
      else
         a = p;
      }
   else {//cba
      c = s;
      a = p;
      }
   }
cout << "a = " << a
     << " b = " << b
     << " c = " << c;
}
```

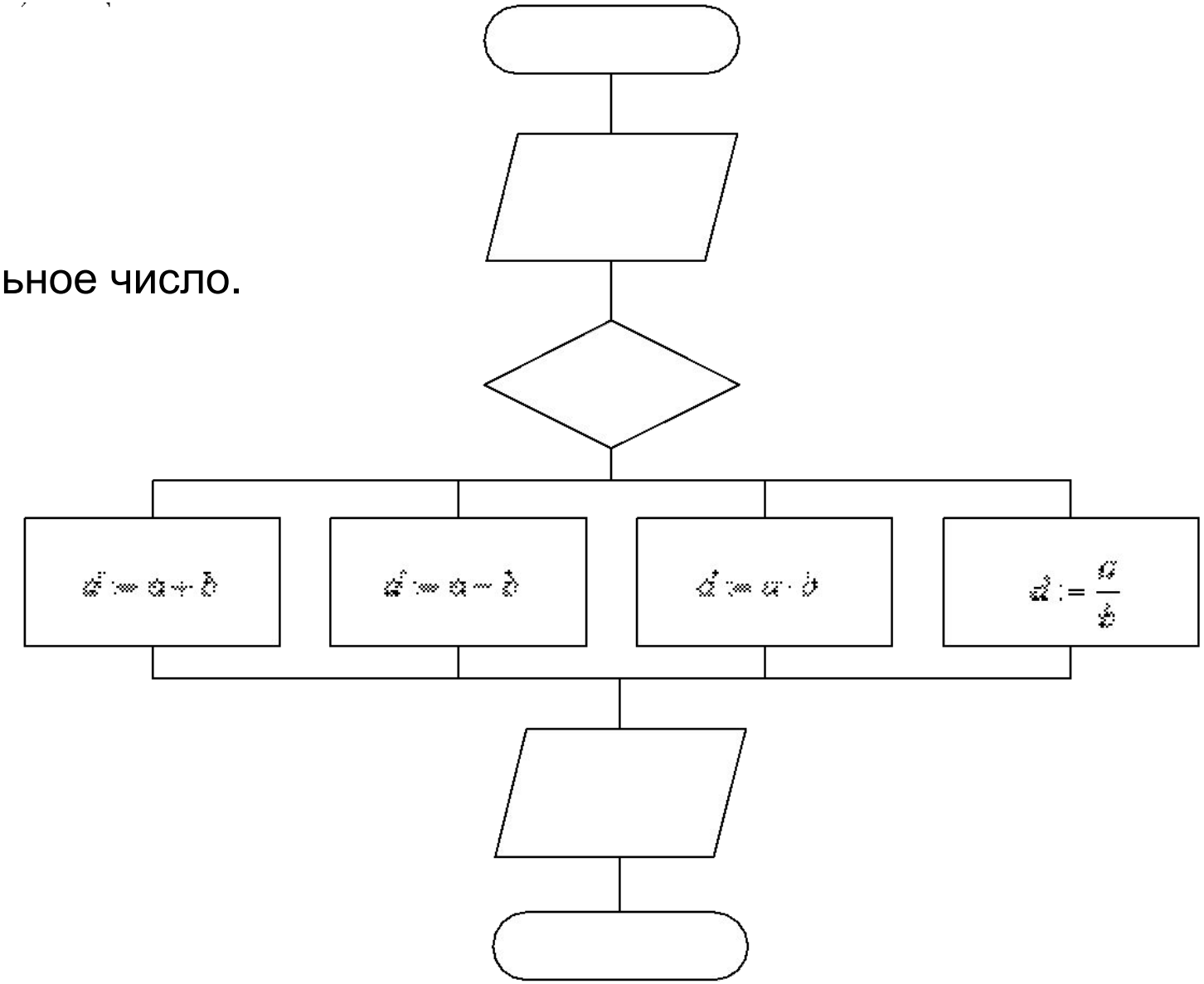
Решение задачи 4

Входные данные:

a, b, – действительные числа;

c – символ (знак операции).

Выходные данные: *d* – действительное число.



```
int main(){
float a, b;
char sign;
//cout << "input 2 float values\n";
cin >> a >> sign >> b;
//cout << "input operation sign: +, -, * or /\n";
switch (sign){
   case '+' : cout << a + b; break;
   case '-' : cout << a - b; break;
   case '*' : cout << a * b; break;
   case '/' : cout << a / b; break;
   default  : cout << "invalid operation sign\n";
   }

}
```