СПбГУТ им. проф. М.А. Бонч–Бруевича
Кафедра программной инженерии и вычислительной техники (ПИ и ВТ)

ПРОГРАММИРОВАНИЕ

Единственный способ изучать новый язык программирования – писать на нем программы.

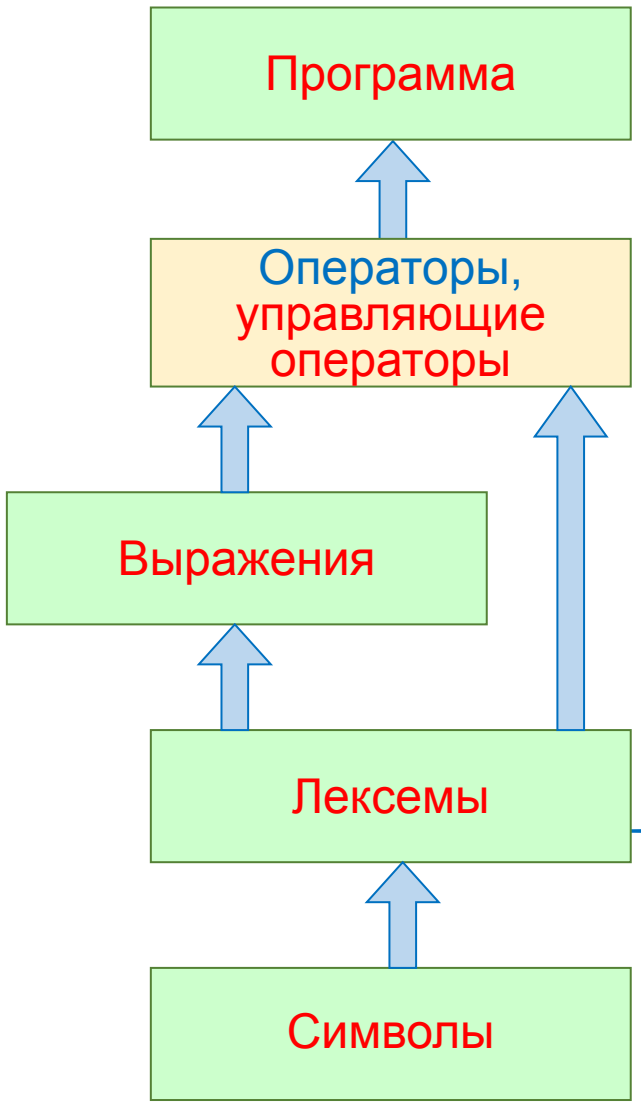
Брайэн Керниган

Лекция 7: *Управляющие инструкции языка Си*

1. Классификация инструкций языка Си
2. Инструкции выбора if, switch
3. Операторы цикла
4. Операторы перехода и возврата

Санкт–Петербург,

Введение. Вспомним состав языка



a=b; for (int i=0;i<n;++i)

a++ - --b/c

- идентификаторы
- ключевые слова
- константы
- знаки операций
- разделители

a-z, A-Z, 0-9, “, {,},|,/,%,…

примеры

1. Классификация инструкций языка Си

<инструкция> ::=
<помеченная-инструкция>
| <инструкция-выражение>
| <составная-инструкция>
| <инструкция-выбора>
| <циклическая-инструкция>
| <инструкция-перехода>

NB: Форма Бэкуса-Наура (БНФ)

::= «Это есть»
| «Или»

- ***Управляющие операторы*** (***инструкции***) – *предназначены для осуществления действий и для управления ходом выполнения программы.*
- Несколько идущих подряд операторов образуют ***последовательность операторов***.
- ***Пустой оператор***
  Самая простая языковая конструкция — это *пустое выражение*, называемое пустым оператором: **;**
- ***Инструкции***
  Инструкция — это некое элементарное действие:
  (выражение);
- ***Блок инструкций***
  Инструкции могут быть сгруппированы в *специальные блоки* следующего вида:

```
{
  (последовательность инструкций);
}
```

- ***Блок инструкций****, также иногда называемый составным оператором*, ограничивается левой фигурной скобкой { в начале и правой фигурной скобкой } — в конце.
  *В **функциях** блок инструкций обозначает тело функции и является **частью определения функции***.
  Также составной оператор может использоваться в операторах **циклов**, **условия** и **выбора**.

Таким образом:

- ***Инструкция*** *или* ***оператор*** *(statement) — наименьшая автономная часть языка программирования; команда или набор команд.*
- ***Оператор*** (***инструкция*** */ statement*) *— это единица выполнения программы.*

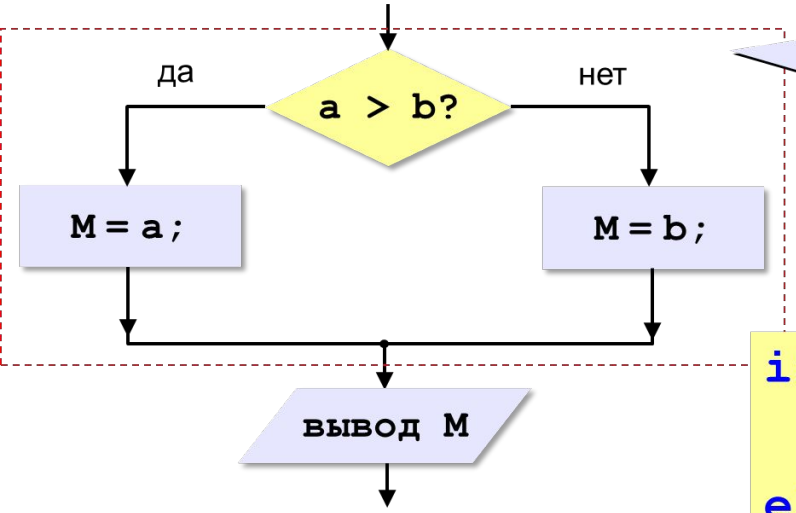
- В языке **Си** любое **выражение**, заканчивающееся символом "точка с запятой" **;** является оператором.
- **Фигурные скобки { }** — это *составной оператор*, является отдельным *блоком* и *в нем можно определять локальные переменные*.
- **Программа** *обычно представляет собой последовательность инструкций.*
- Многие языки программирования (например, **Си**) различают **инструкцию** и **определение**:
  - различие в том, что **инструкция** *исполняет код*, а **определение** *создаёт идентификатор* (то есть можно рассматривать **определение** *как инструкцию присваивания*).

2. Инструкции выбора if, switch

```
<инструкция-выбора> ::=
     'if' '(' <выражение> ')' <инструкция>
 |   'if' '(' <выражение> ')' <инструкция> 'else'
<инструкция>
 |   'switch' '(' <выражение> ')' <инструкция>
```

Условный оператор

Задача: изменить порядок действий в зависимости от выполнения некоторого условия.



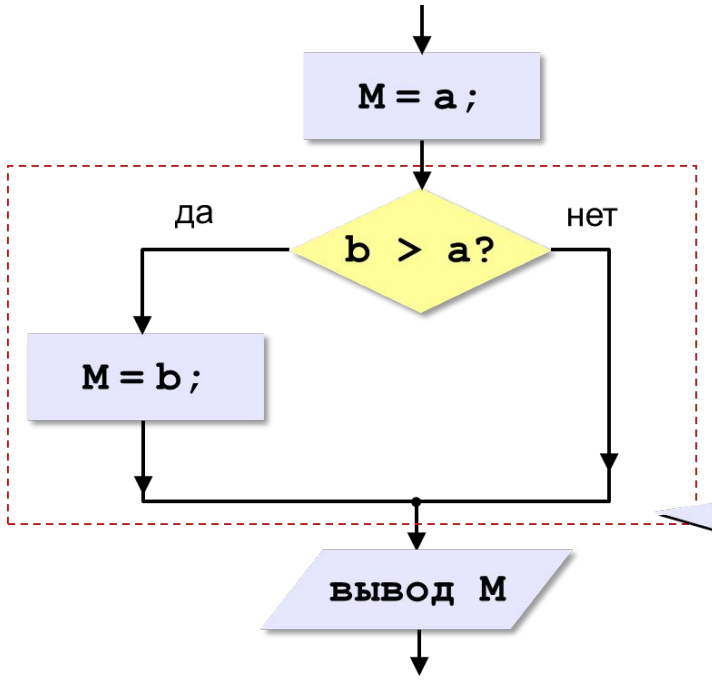
полная форма ветвления

? Если a = b?

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

```
if (выражение)
оператор_1;
   else оператор_2;
```

Условный оператор: неполная форма



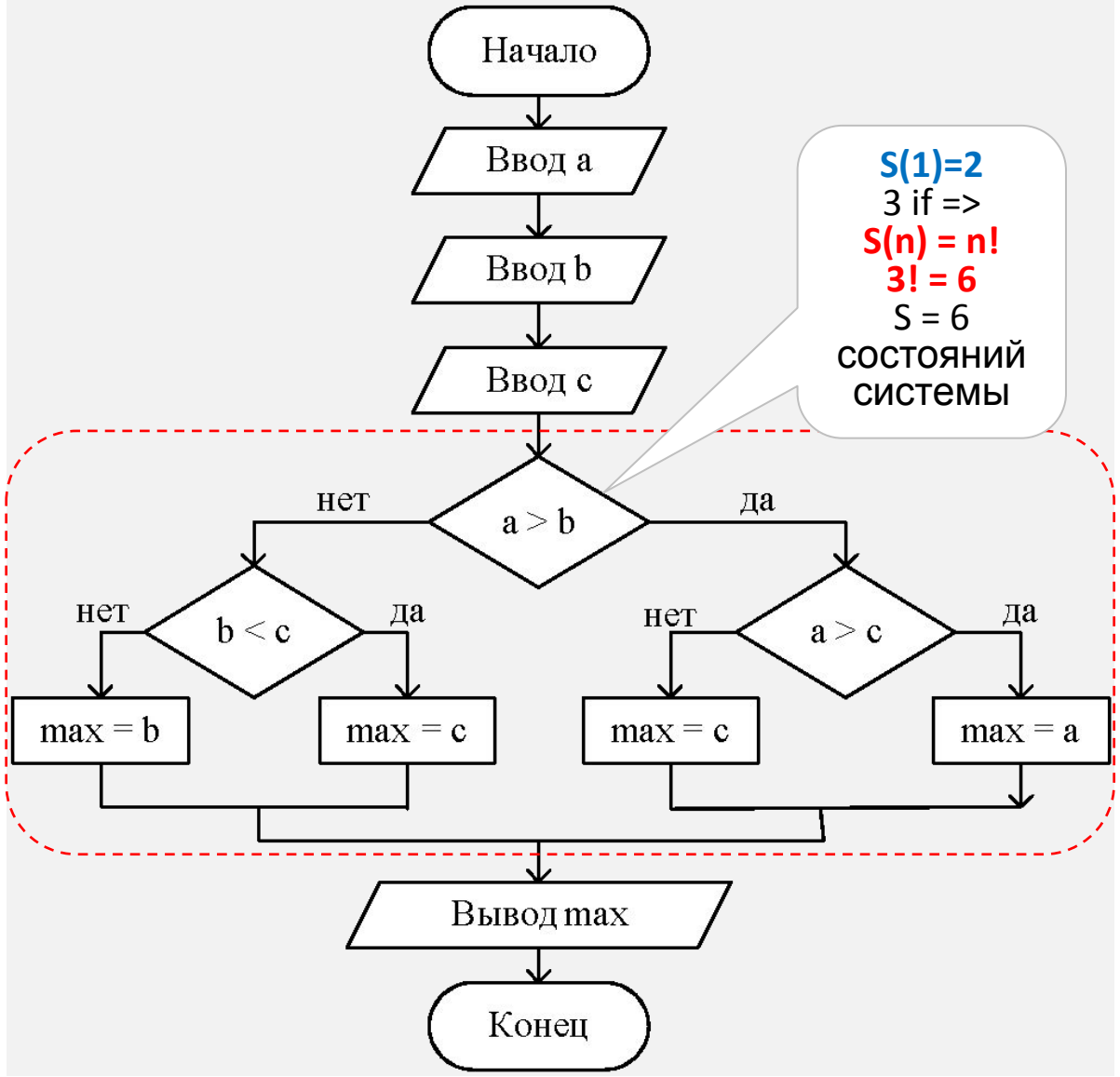

```
M = a;
if ( b > a )
   M = b;
```

неполная форма ветвления

```
if (выражение)
оператор_1;
```

Инструкции выбора if

На блок-схеме алгоритма (поиск max):



Примеры использования:

```
if (x>0) j=k+10;
   else m=1+10;
```

```
if(n>0)   //n=5, z=0,a=1,b=2, z-?          Z=2
  if(a>b)  z=a;
    else  z=b;
```

```
if(n>0) //n=5, z=0,a=1,b=2, z-?            Z=0
 {if(a>b)  z=a; }
  else  z=b;
```

Но!..

- Вы увеличите сложность с любым новым условным требованием, реализованным с помощью If-else.
- Логика ветвления в примере не очень сложна — но попробуйте добавить новые условия, и вы увидите, что она усложняется в разы!..
- Если в вашем коде слишком много условных операторов, то код очень сложно проследить и еще сложнее модифицировать.
- Когда нам нужно внести какое-нибудь изменение, то придется пройтись по нескольким частям кода. При добавлении нового функционала нужно модифицировать ранее написанный код, чтобы этот новый функционал ужился с имеющимся кодом.
- Еще часто бывает так, что по всей базе кода разбрызганы дубли одной и той же логики. Поэтому когда нам нужно что-то поменять, данное изменение надо будет вносить много раз в самых разных частях кода, причем ни в коем случае не забыв ни одного блока.

Организация множественного выбора

```
if (выражение1) оператор_1;
   else if (выражение2) оператор_2;
         else if (выражение3) оператор_3;
        else оператор_4;
```

- Если **не** используются **фигурные скобки**, то else соответствует ближайшему if
- Любое количество конструкций else-if
  - ```
    if  (n<0) printf (“ отрицательное\n”);
       else if (n==0) printf (“нулевое\n”);
      else printf (“положительное\n”);
    ```
  - ```
    char ZNAC;
     int x,y,z;
       if (ZNAC == '-') x = y - z;
          else  if (ZNAC == '+') x = y + z;
             else if (ZNAC == '*') x = y * z;
                else if (ZNAC == '/') x = y / z;
                   else ...
    ```

Символ Кронекера — индикатор равенства элементов, формально: функция двух целых переменных, которая равна 1, если они равны, и 0 в противном случае:

$$\delta_{ij} = \begin{cases} 1, & i = j, \\ 0, & i \neq j. \end{cases}$$

Тернарная условная операция: `c = a > b ? a : b;`
//присвоить с максимальное

- Реализованная во многих языках программирования операция, возвращающая свой второй или третий операнд в зависимости от значения логического выражения, заданного первым операндом.
- Данный оператор порой бывает удобен в случаях, если при выполнении какого-то условия или его невыполнении нам нужно присвоить строго определённое значение какой-либо переменной.
- Хотя такая ситуация и не столь частая, но в случае её возникновения тернарный оператор намного удобнее и читабельнее нежели конструкция IF.
- Аналогом тернарной условной операции в математической логике и булевой алгебре является условная дизъюнкция, которая записывается в виде [p,q,r] и реализует алгоритм: «**если q, то p, иначе r**»

```
int n, m, res;
printf("Please enter an integer (n)\r\n");
scanf("%d", &n);
printf("Please enter an integer (m)\r\n");
scanf("%d", &m);
res = (n > m) ? n : m;
printf("The maximum number of entered: %d", res);
```

Символ Кронекера: `y = x == 0 ? 1 : 0;`

Минимальное из чисел a и b: `min = (a < b) ? a : b;`

Ветвления

Знаки отношений:

`>` `<`	больше, меньше
`>=`	больше или равно
`<=`	меньше или равно
`==`	равно
`!=`	не равно

Оператор if может быть вложенным:

```
#include <stdio.h>
int main()
{
 int key;  // объявляем целую переменную key
 printf("Введите номер пункта, 1 или 2: ");
 scanf("%d", &key);   // вводим значение переменной key
 if (key == 1)       // если key = 1
   printf("\n Выбран первый пункт"); // выводим сообщение
 else if (key == 2)   // иначе если key = 2
   printf("\n Выбран второй пункт"); // выводим сообщение
 else                 // иначе
  printf("\n Первый и второй пункты не выбраны"); /* выводим
                                                    сообщение
*/
 getchar();
 return 0;
}
```

```
Enter the item number, 1 or 2: 1

The first item is selected
```

- При использовании вложенной формы оператора **if** опция **else** связывается с последним оператором **if**.
- Если требуется связать опцию **else** с предыдущим оператором **if**, внутренний условный оператор заключается в **фигурные скобки**

Сложные условия

Пример: набор сотрудников в возрасте 25-40 лет (включительно).

```
if ( v >= 25 && v <= 40 )
  printf("подходит");
else
  printf("не подходит");
```

&& «И» || «ИЛИ» ! «НЕ»

- ***Сложное условие*** *– состоит из двух или нескольких простых отношений (условий), которые объединяются с помощью логических операций.*
- Запись логических связок на языке **Си**:
  - `&&` – логическое умножение (**И**);
  - `||` – логическое сложение (**ИЛИ**);
  - `!` – логическое отрицание (**НЕ**).
- **Логическое умножение** (операция И) требует одновременное выполнение двух условий:
  - условие_1 && условие_2
  - будет принимать истинное значение, только если оба простых условия истинны одновременно.

Порядок выполнения сложных условий:

- Выражения в скобках
- `!` («НЕ», отрицание)
- Отношения (`<`, `>`, `<=`, `>=`)
- `==`, `!=` (равно/не равно)
- `&&` («И»)
- `||` («ИЛИ»)
- Для изменения порядка действий используются круглые скобки

Множественный выбор. Инструкция switch

Инструкция **switch** имеет следующий вид:

```
switch (выражение для сравнения)
{
 case совпадение1: команда; break;
 case совпадение2: команда; break;
 case совпадение3: команда; break;
 ……..
 default: оператор; break;
}
```

- Текст **default**: инструкции может отсутствовать
- Порядок работы
  - Вычисляется выражение в скобках, результат приводится к **int**
  - Если значение совпадает со значением одного из выражений после **case**, то управление передаётся на первую инструкцию после соотв. двоеточия.
  - Дальнейшая работа зависит от этих инструкций
  - Иначе управление передается на первую инструкцию после **default:**
- В отличие от операторов **if-else**, оператор **switch** применим к известному числу возможных ситуаций.
- Можно использовать простые типы **byte**, **short**, **char**, **int**.
- Также можно использовать **Enum**
- Каждая секция case обычно заканчивается командой **break**, которая передаёт управление к концу команды **switch**. Если не использовать **break**, выполнение кода продолжится.
- Дублирование значений **case** не допускается. Тип каждого значения должен быть совместим с типом выражения.

Пример:

```
if (m == 1) printf("\n январь");
if (m == 2) printf("\n февраль");
...
if (m == 12) printf("\n декабрь");
```

```
switch (m)
{
  case 1: printf("\n январь");
        break;
  case 2: printf("\n февраль");
        break;
  ...
  case 12: printf("\n декабрь");
        break;
  default: printf("\n Ошибка");
  }
```

```
// Если не ставить break:
switch ( m )
{
  case 1:     printf("\n январь");
  case 2:     printf("\n февраль");
  case 3:     printf("\n март");
  default:    printf("\n Ошибка");
 }
```

При m = 2:

февральмартОшибка

Множественный выбор. Инструкция switch

Упрощенное суждение:

- ✔ Инструкция **if** позволяет сделать выбор между двумя ветвями выполняемыми программы.
- ✔ Инструкция **switch** - это инструкция многонаправленного ветвления, которая позволяет выбрать одну из множества альтернатив.

Как обычно поясняется работа **switch**'а в учебниках:

```
switch (x)
{
  case 10:
    y = 1;
    break;
  case 11:
  case 12:
    y = 2;
    break;
  default:
    y = 3;
    break;
}
```

```
if (x == 10)
  {
    y = 1;
  }
else if (x == 11 || x == 12)
  {
    y = 2;
  }
else
  {
    y = 3;
  }
```

- С точки зрения исполнения программы эти примеры действительно эквивалентны.
- Более того, большинство компиляторов скорее всего построят для этих примеров один и тот же код (или очень близкий).

Как на самом деле работает **switch**:

- ❑ Оператор **switch** на самом деле является коммутируем переходом.
- ❑ Т.е. в зависимости от значения ключа (то, что стоит в скобках после слова **switch**) происходит переход на ту или иную метку, а операторы case определяют эти самые метки.
- ❑ Посмотрим это на примере, содержащем в том числе провалы (отсутствие **break**'ов).

```
switch (x)
{   // <--- скобка 1
  case 10:
    y = 1;
    break;
  case 11:
  case 12:
    y = 2;
    /* break; */  <--- тут уберём break и устроим провал
  default:
    y = 3;
    break;
}    // <--- скобка 2
```

Семантическим эквивалентом данному коду будет код представленный на следующем слайде…

Инструкция switch vs if

Семантическим эквивалентом **switch** -коду будет код представленный ниже:

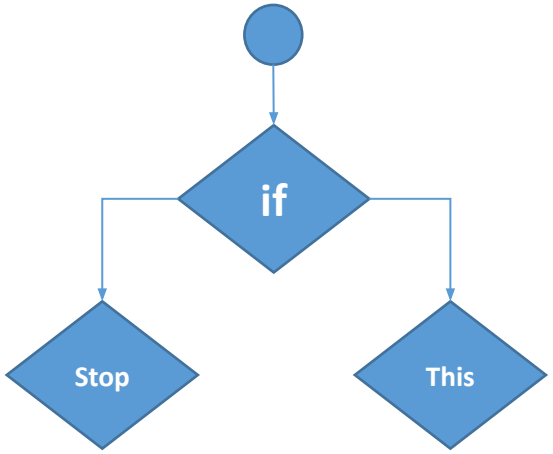
```
/* Данный набор операторов if и goto есть семантический эквивалент
 * оператора switch (x) */
if (x == 10)
  goto L10;
else if (x == 11)
  goto L11;
else if (x == 12)
  goto L12;
else
  goto Ldefault;
{                   <--- скобка 1
L10:                <--- метка "case 10"
  y = 1;
  goto Finish;          <--- break
L11:                <--- метка "case 11"
L12:                <--- метка "case 12"
  y = 2;
  /* goto Finish; */    <--- закомментированный break
Ldefault:           <--- метка "default"
  y = 3;
  goto Finish;          <--- break
}                   <--- скобка 2
Finish:                 <--- метка за пределами switch'а, куда ведут все break'и
                        эта метка полностью эквивалентна тому, что есть
                        в циклах for и while
```

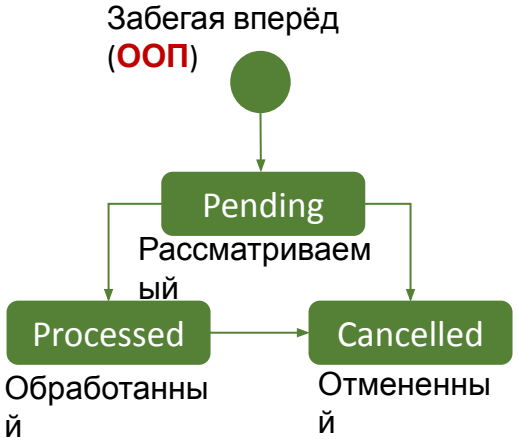
- Если посмотреть на пример пояснения из учебников и рассмотренный пример, то с виду кажется, что принципиальных отличий одного от другого нет и второе элементарно сводится к первому.
- Конкретно в данном случае это действительно так.
- Вместо длинной цепочки из N if'ов мы имеем очень коротенький код по динамическому переходу (переходу на заранее неизвестный адрес).
- Важно понимать, что скорость исполнения этого кода НЕ зависит от размеров таблицы: т.е. switch из 1000 элементов работает с такой же скоростью, как switch из 5 элементов (чего не скажешь о цепочке if'ов).
- В то время как массив с метками переходов является статически инициализированным (т.е. в бинарном файле лежит уже заполненная таблица, которую в run-time **не надо перевычислять**).
- Есть **некоторые тонкие моменты**, остающихся на усмотрение компилятора.
- Например, если в switch'е мы имеем всего две альтернативы 1 и 1 000 000, то таблица, построенная по такому принципу, занимала бы **миллион слов** (4 мегабайта в 32-битном режиме).
- Компилятор в таком случае вместо динамического перехода построит цепочку из двух if'ов.

Как бороться с новыми условиями?..

Если новая функция обычно была реализована с помощью некоторой условной проверки, то вы можете просто создать **новый объект** (структуру/класс) **состояния**.

Это очень просто. Вам больше не придется иметь дело с громоздким оператором if-else.





Рефакторинг ветвящейся логики из нашего кода представляет собой трехэтапный процесс:

1) Создать абстрактный класс базового состояния
2) Реализовать каждое состояние как отдельный класс, наследующийся от базового состояния
3) Класс имеет приватный или внутренний метод, который принимает класс базового состояния в качестве параметра

Созданный класс — это делегирование полномочий осуществления принятия и отклонения.

```
for(int i = 0; i < 10; i++)
   for(int j = 0; j < 10; j++)
      if(...)
      {} else {}
```

- **Алгоритмическая сложность O(n)** будет: **i * j * 2** ? (*самый простой случай*)
- Итоговая сложность будет зависеть от того, что происходит внутри **if**
- Сложность **i * j** набирается только из двух циклов
- Общая сложность будет произведением:
  i * j * (сложность того, что в if)

Варианты рефакторинга

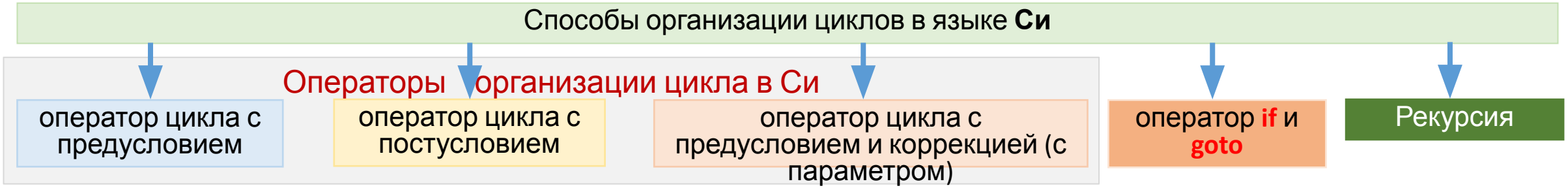
- Тернарный оператор: **z = (x > y) ? x : y;**
- Составные условия для выполнения блока под условным оператором. Но это **усложняет логику**!
- Как вариант рефакторинга, можно переделать выражение в **функцию**, чтобы код стал более читаемым.
- **Switch**. Наиболее очевидной заменой "if" являются высказывания-переключатели.
- Логические операторы (**&&** и **||**).
- …

3. Операторы цикла (for, while, do-while)

```
<циклическая-инструкция> ::=
      'while' '(' <выражение> ')' <инструкция>
|     'do' <инструкция> 'while' '(' <выражение> ')'
|     'for' '(' [<выражение>] ';' [<выражение>] ';'
[<выражение>] ')'
            <инструкция>
```

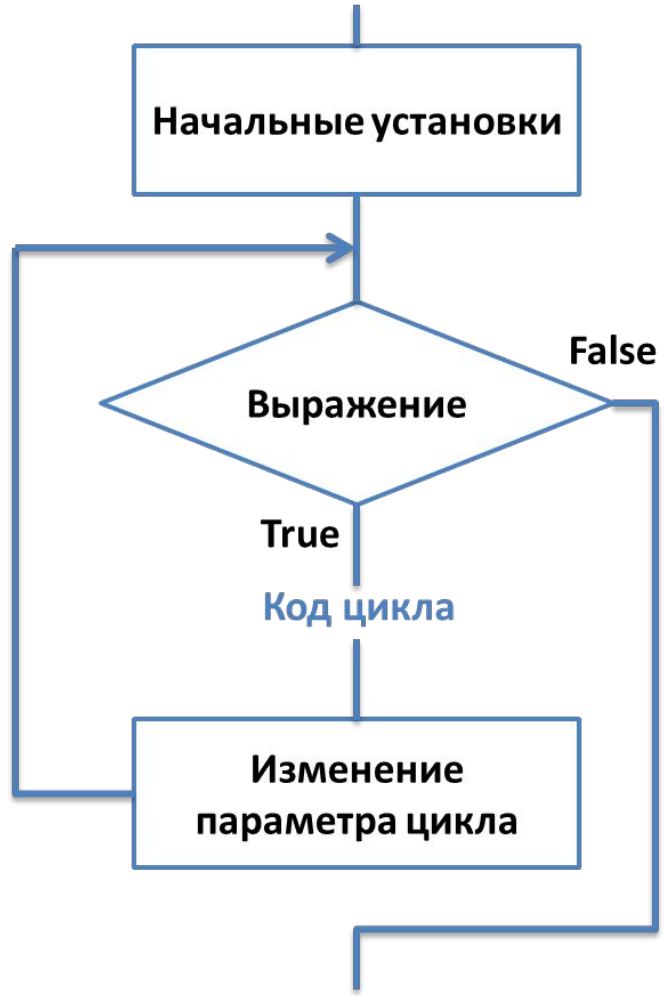
- В **цикле** for любое из выражений может отсутствовать
- **Циклы** с использованием:
  - оператора цикла с предусловием **while**
  - оператора цикла с постусловием **do while**
  - оператора цикла с параметром **for**
- **Итерация** (**шаг**) – *один проход цикла.*

- ***Цикл*** *– это организованное повторение некоторой последовательности операторов.*
- Это одно из фундаментальных понятий программирования.
- Любой цикл состоит из **кода цикла**, т.е. тех операторов, которые выполняются несколько раз, начальных установок, модификации параметра цикла и проверки условия продолжения выполнения цикла
- Один проход цикла называется **шагом** или **итерацией**. Проверка условия продолжения цикла происходит на каждой итерации либо до выполнения кода цикла (**с предусловием**), либо после выполнения (**с постусловием**)



Оператор с предусловием while

```
while (выражение)
код цикла;
```



- ***Выражение*** определяет условие повторения **кода цикла**, представленного простым или составным оператором
- ***Код цикла*** может включать любое количество операторов, связанных с конструкцией **while**, которые нужно заключить в **фигурные скобки** (организовать **блок**), если их более одного
- **Переменные**, **изменяющиеся в коде цикла** и используемые при проверке условия продолжения, называются ***параметрами цикла***.
- **Целочисленные параметры цикла**, изменяющиеся с постоянным шагом на каждой итерации, называются ***счетчиками цикла***
- ***Начальные установки*** могут явно не присутствовать в программе, их смысл состоит в том, чтобы до входа в цикл задать значения переменным, которые в этом цикле используются
- ***Цикл завершается***, если условие его продолжения не выполняется.
- Возможно **принудительное завершение** как **текущей итерации**, так и **цикла в целом**:
  - **continue** – переход к следующей итерации цикла
  - **break** – выход из цикла.

Пример: *Организации паузы в работе программы с помощью цикла, выполняющегося до тех пор, пока не нажата любая клавиша*:

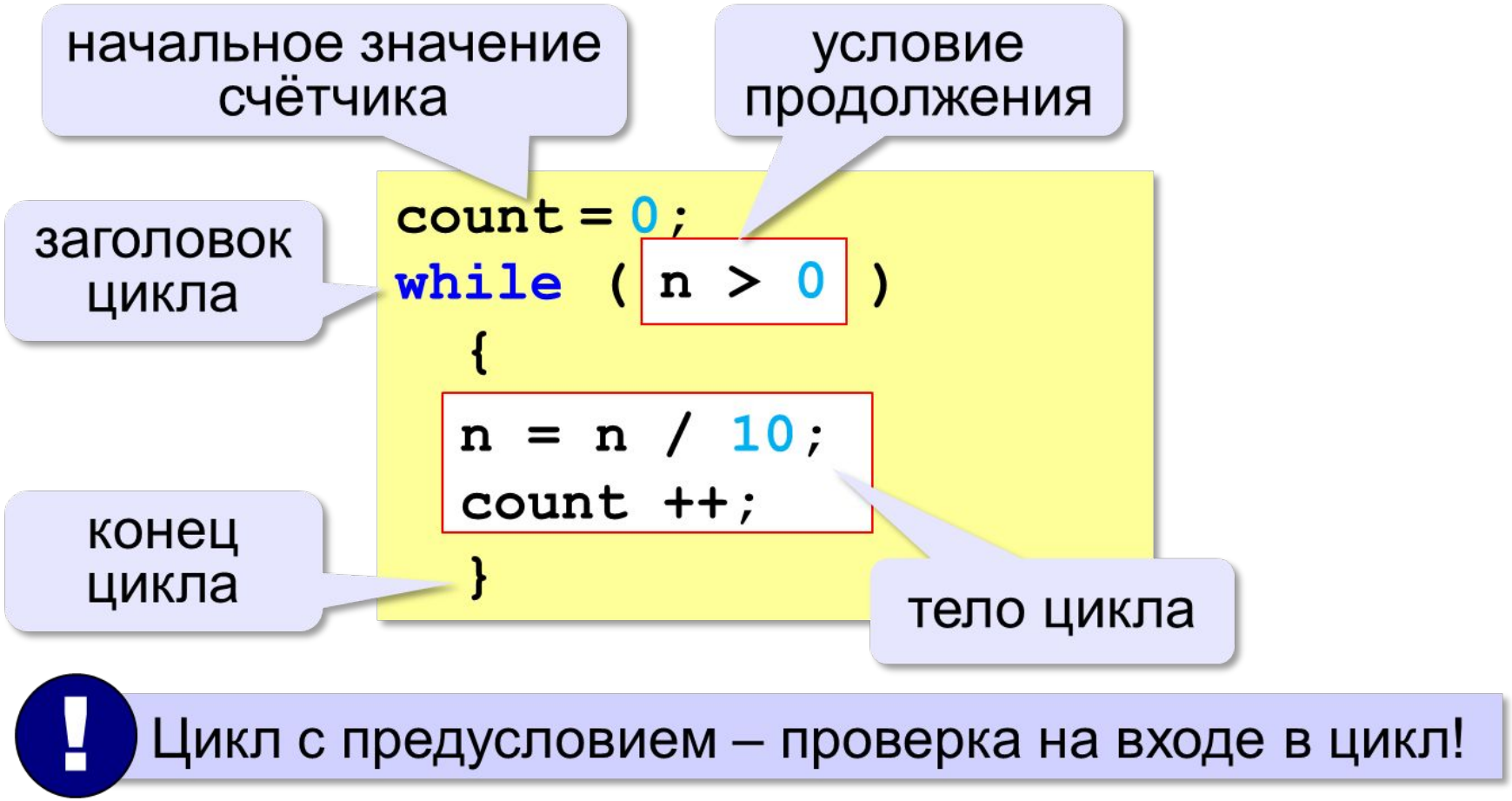
```
while (!kbhit());
```

Функция **kbhit()** возвращает значение **> 0**, если нажата любая клавиша.
Функция **getch()** возвращает код нажатой клавиши.

Пример: *Организация выхода из бесконечного цикла по нажатии клавиши* **Esc** (код 27):

```
while (1)
{       // Бесконечный цикл
```

Цикл с предусловием while



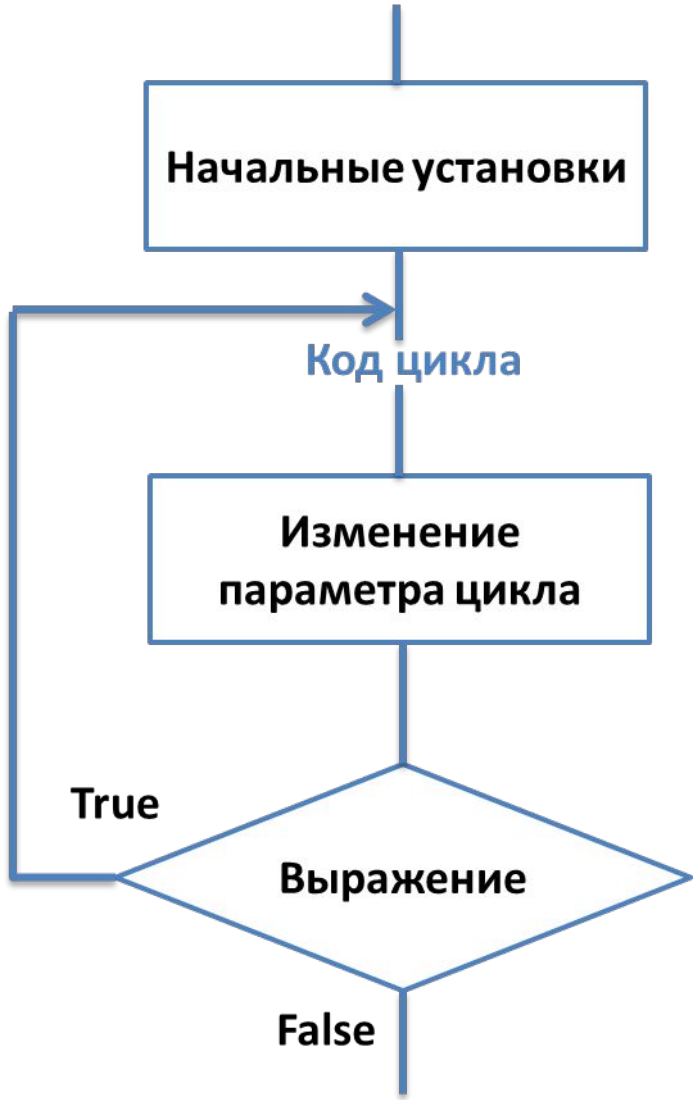
Цикл с предусловием – проверка на входе в цикл!

Оператор цикла с постусловием do – while

Общий вид записи:

```
do
  код цикла;
while (выражение);
```

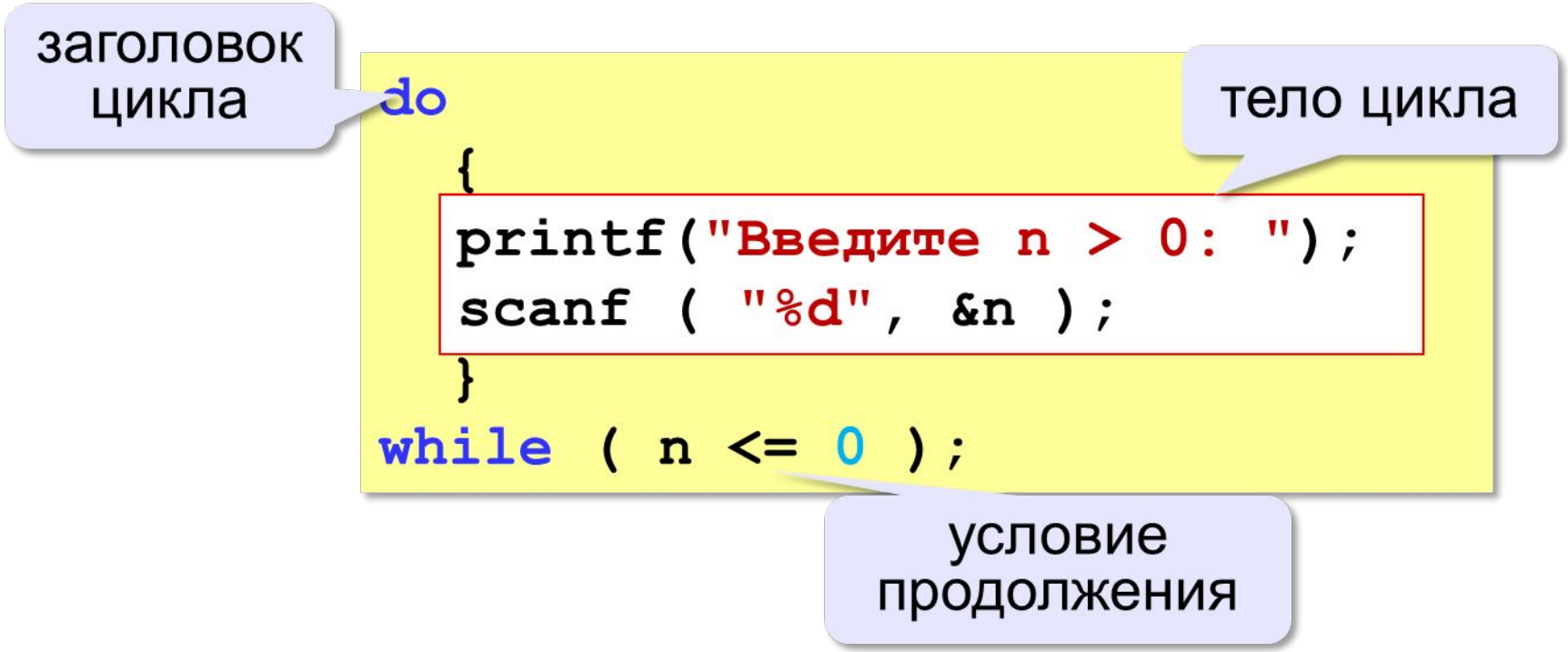
- **Выражение** определяет условие повторения **кода цикла**, представленного простым или составным оператором
- **Код цикла** будет выполняться до тех пор, пока **выражение истинно.**
- Данный цикл всегда **выполняется хотя бы один раз**, даже если изначально выражение ложно.
- У цикла **do-while** условие указывается после тела цикла.



Пример:

```
char answer;
do
{
  puts(" Продолжить работу (y/n) ? ");
  scanf(" %c ", &answer);
}
while ((answer=='y')||(answer=='Y'));
```

Цикл с постусловием do – while



- при входе в цикл условие ***не*** **проверяется**
- цикл всегда выполняется **хотя бы один раз**

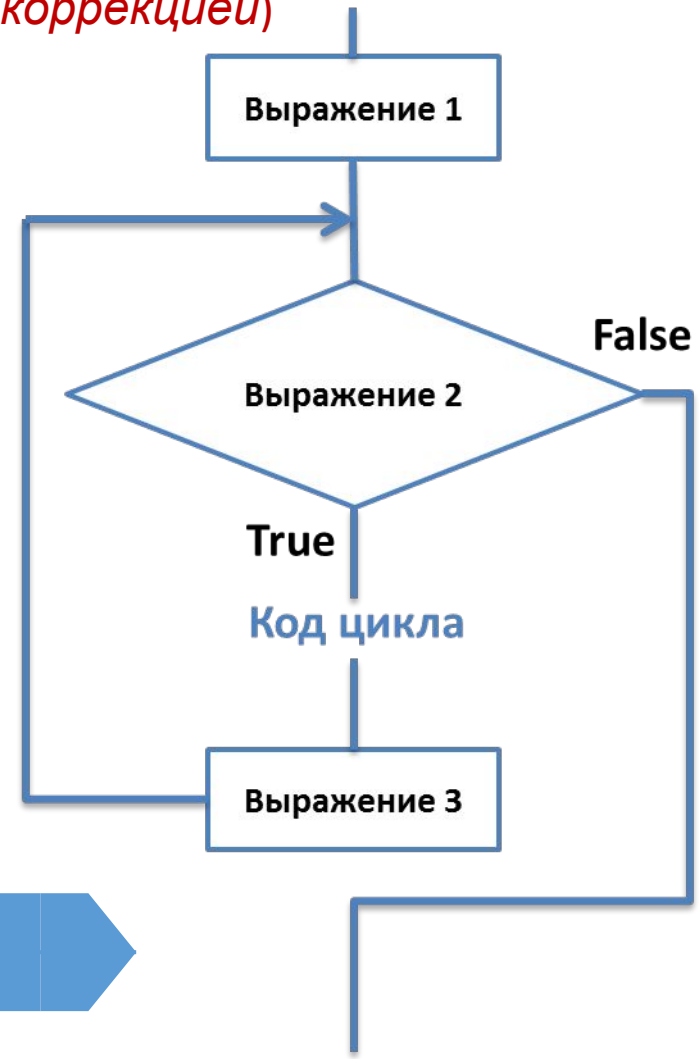
Оператор цикла с параметром for (*с предусловием и коррекцией*)

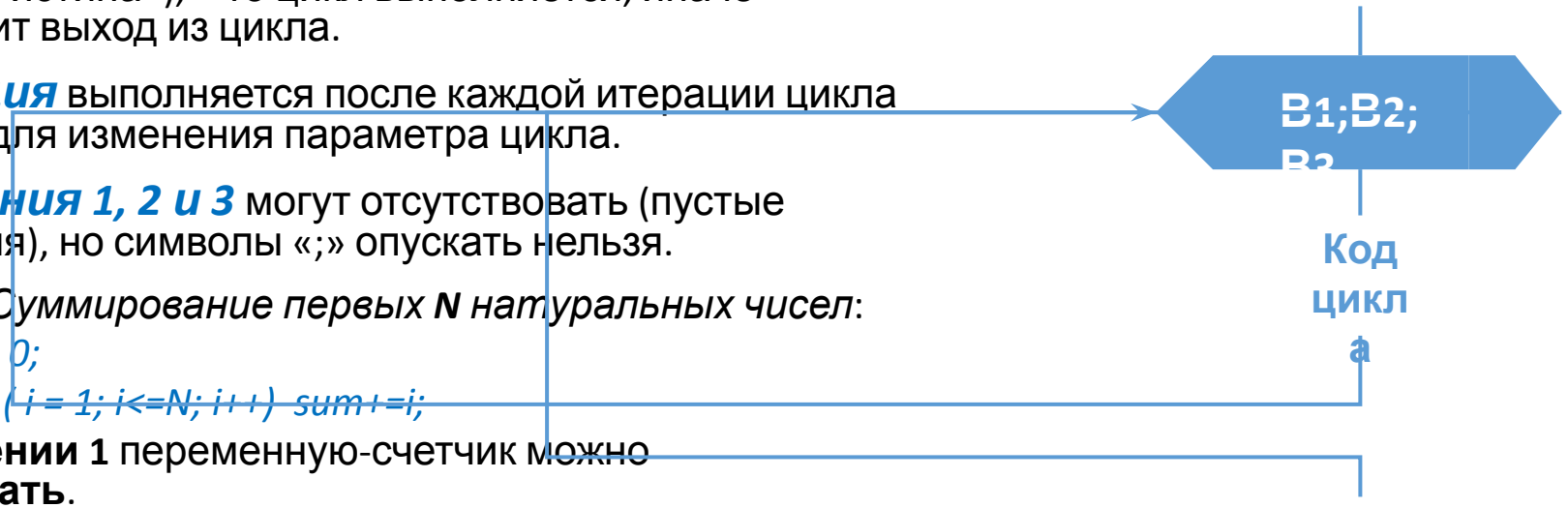
Общий вид оператора:

for (выражение 1; выражение 2; выражение 3)

код цикла; // тело цикла

- ***выражение 1*** – инициализация счетчика (параметр цикла)
- ***выражение 2*** – условие продолжения счета
- ***выражение 3*** – коррекция счетчика
- ***Инициализация*** используется для присвоения счетчику (параметру цикла) начального значения.
- ***Выражение 2*** определяет условие выполнения цикла. Как и в предыдущих случаях, если его результат не нулевой («истина»), – то цикл выполняется, иначе – происходит выход из цикла.
- ***Коррекция*** выполняется после каждой итерации цикла и служит для изменения параметра цикла.
- ***Выражения 1, 2 и 3*** могут отсутствовать (пустые выражения), но символы «;» опускать нельзя.





Пример: *Суммирование первых **N** натуральных чисел*:

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

В **выражении 1** переменную-счетчик можно **декларировать**.

```
for (int i = 1; i<=N; i++)
```

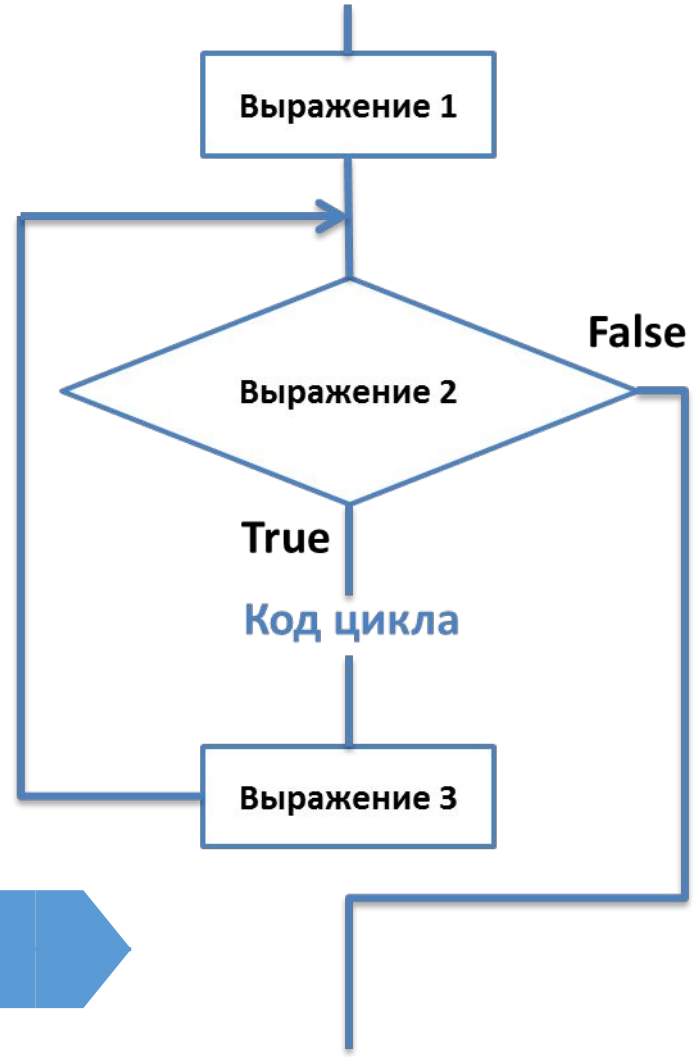
Оператор цикла с параметром for

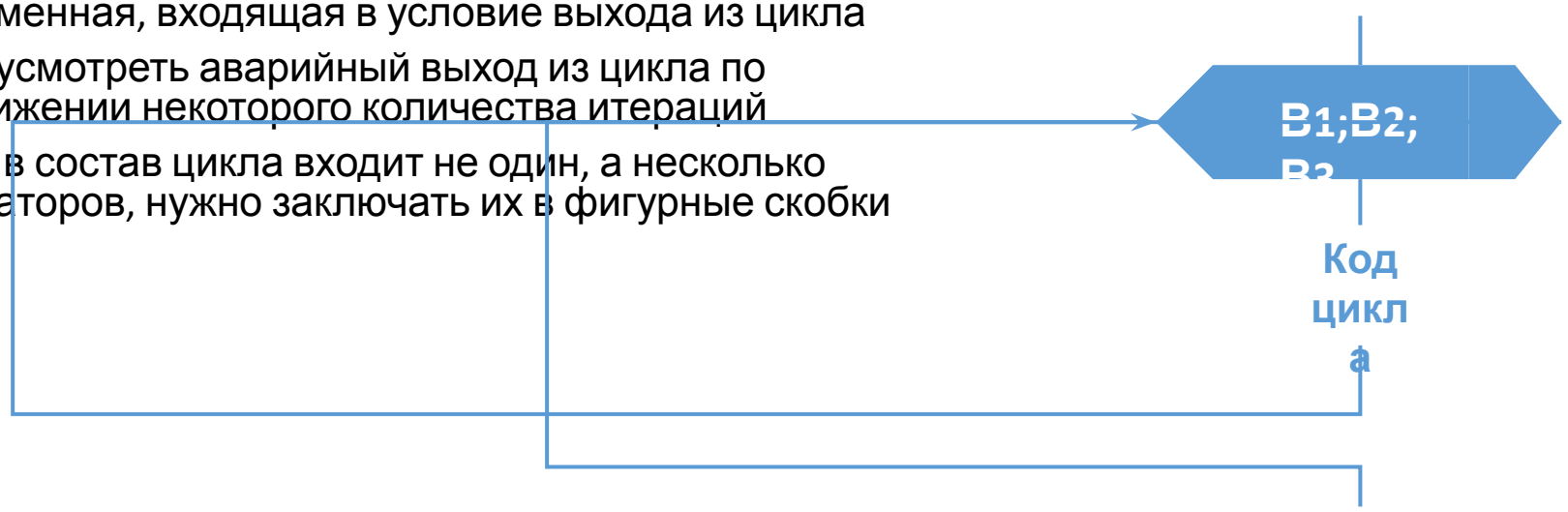
Общий вид оператора:

for (выражение 1; выражение 2; выражение 3)

код цикла; // тело цикла

- Наиболее часто встречающиеся **ошибки** при создании циклов – это **использование в коде цикла** неинициализированных переменных и неверная запись условия выхода из цикла
- Чтобы избежать **ошибок**, нужно:
  - проверить, всем ли переменным, встречающимся в правой части операторов присваивания в коде цикла, присвоены до этого начальные значения (а также возможно ли выполнение других операторов)
  - проверить, изменяется ли в цикле хотя бы одна переменная, входящая в условие выхода из цикла
  - предусмотреть аварийный выход из цикла по достижении некоторого количества итераций
  - если в состав цикла входит не один, а несколько операторов, нужно заключать их в фигурные скобки





Оператор цикла с параметром for

Задача. Вывести все степени двойки от 2^1 до 2^{10}.

Можно ли сделать с циклом «пока»?

```
k = 1;
n = 2;
while ( k <= 10 )
   {
   printf("%d\n", n);
   n *= 2;
   k ++;
   }
```

```
n = 2;
for( k=1; k<=10; k++ )
   {
   printf("%d\n", n);
   n *= 2;
   }
```

цикл с переменной

Вложенные циклы

```
for ( n=2; n<=1000; n++ )
  {
  count=0;
  for ( k=2; k<n; k++ )
      if ( n%k==0 )
        count++;
  if ( count == 0 )
    printf("%d\n", n);
  }
```

вложенный цикл

4. Операторы перехода и возврата *break, continue, goto, return*

```
<инструкция-перехода> ::=
      'goto' <идентификатор> ';'
|     'continue ';'
|     'break' ';'
|     'return' [<выражение>] ';'
```

- **continue ;**
  - Передаёт управление на проверку условия в **while** и **do-while** и на вычисление третьего выражения в **for**
  - Разрешено только в операторах цикла
- **break ;**
  - Передаёт управление на первый оператор после цикла или после оператора выбора
  - Разрешено в циклах и в операторе выбора **switch**
- **return выражение ; и return ;**
  - Завершает работу текущей функции и возвращает управление вызывающей функции
  - выражение должно быть приводимым к типу результата функции с помощью стандартных преобразований
- **goto идентификатор ;**
  - Передаёт управление на оператор, помеченный меткой идентификатор
  - Рекомендуется передавать управление только вперёд про тексту программы
  - Разрешено передавать управление из блока { } наружу за исключением выхода из функции

goto идентификатор ; // продолжение

- Нет смысла (но не запрещено) передавать управление внутрь блока { }
- После такой передачи управления значения переменных, описанных внутри { }, неопределенны
- идентификатор должен быть меткой инструкции

Метка представляет собой идентификатор, оформленный по всем правилам идентификации переменных с символом «двоеточие» после него, например, пустой помеченный меткой m1 оператор:

```
m1:   /* … */;
```

Область действия **метки** – функция, где эта метка определена. В случае необходимости можно использовать **блок**.

Операторы и функции передачи управления

Циклы и **переключатели** можно ***вкладывать вдруг в друга*** и наиболее характерный оправданный случай использования оператора **goto** – выполнение прерывания (организация выхода) во вложенной структуре.

Например, при возникновении грубых неисправимых ошибок необходимо выйти из двух (или более) вложенных структур (где нельзя использовать непосредственно оператор **break**, т.к. он прерывает только самый внутренний цикл)

```
for (...)
        for (...) {
...
            if (ошибка)  goto error;
        }
                ...
error:  операторы для устранения ошибки;
```

- Оператор **goto** можно использовать для организации переходов из нескольких мест функции в одно, например, когда перед завершением работы функции необходимо сделать одну и ту же операцию
- Оператор **continue** может использоваться во всех типах циклов (но не в операторе-переключателе **switch**).

Наличие **оператора** continue вызывает пропуск «оставшейся» части итерации и переход к началу следующей, т.е. досрочное завершение текущего шага и переход к следующему шагу

- В циклах **while** и **do-while** это означает непосредственный переход к проверочной части.
- В цикле **for** управление передается на шаг коррекции, т.е. модификации **выражения 3**.
- Оператор **continue** часто используется, когда последующая часть цикла оказывается слишком сложной, так что рассмотрение условия, обратного проверяемому, приводит к слишком высокому уровню вложенности программы.
- Оператор **break** производит досрочный выход из цикла или оператора-переключателя **switch**, к которому он принадлежит, и передает управление первому оператору, следующему за текущим оператором.
  То есть **break** обеспечивает переход в точку кода программы, находящуюся за оператором, внутри которого он (**break**) находится
- Оператор **return** производит досрочный выход из текущей функции.
  Он также возвращает значение результата функции:
  return выражение;
  Выражение должно иметь скалярный тип
- Функция **exit** выполняет прерывание программы и используется для нормального, корректного завершения работы программы при возникновении какой-либо внештатной ситуации, например, ошибка при открытии файла. При этом записываются все буферы в соответствующие файлы, закрываются все потоки и вызываются все зарегистрированные стандартные функции завершения.

Прототип этой функции приведен в заголовочном файле ***stdlib.h*** и выглядит так: ***void exit (int exit_code);***

Параметр данной функции – ненулевое целое число, передаваемое системе программирования (служебное сообщение о возникшей внештатной ситуации).

Для завершения работы программы также может использоваться функция ***void abort (void);***

Рекомендации по программированию

Выражение, стоящее в круглых скобках операторов *if*, *while* и *do-while*, вычисляется по правилам стандартных приоритетов операций

Если в какой-либо ветви вычислений условного оператора или в цикле требуется выполнить два (и более) оператора, то они при помощи фигурных скобок объединяются в блок

Проверка вещественных величин на равенство, как правило, из-за ограниченной разрядности дает неверный результат

Чтобы получить максимальную читаемость и простоту структуры программы, надо правильно выбирать способ реализации ветвлений (с помощью *if*, *switch*, или условных операций), а также наиболее подходящий оператор цикла

Выражение в операторе *switch* и константные выражения в *case* должны быть целочисленного или символьного типов

Рекомендуется использовать в операторе *switch* ветвь *default*

После каждой ветви для передачи управления на точку кода за оператором *switch* используется оператор *break*

При построении любого цикла надо не забывать тот факт, что в нем всегда явно или неявно присутствуют четыре основных элемента: начальные установки, код цикла, модификация параметра цикла и проверка условия на продолжение цикла

Если количество повторений цикла заранее не известно (реализуется итерационный процесс), необходимо предусмотреть аварийное завершение цикла при получении достаточно большого количества итераций

При использовании бесконечного цикла обязательно необходима организация выхода из цикла по условию

ЛИТЕРАТУРА

1. Демидович Е. Основы алгоритмизации и программирования. Язык Си: учебное пособие – СПб.: БХВ – Петербург, 2006. – 440с.
2. Жешке Р. Толковый словарь стандарта языка Си. – СПб.: Питер, 1994. – 221с.
3. Керниган Б., Ритчи Д. Язык программирования Си: Пер. с англ. – 2-е изд., перераб. и доп. – М.: Финансы и статистика, 1992. – 272с.
4. Кочан С. Программирование на языке С, 3-е издание: Пер. с англ. – М.: ООО “И. Д. Вильямс”, 2007. – 496с.
5. Подбельский В., Фомин С. Программирование на языке Си: учебное пособие. 2-е доп. Изд. – М: Финансы и статистика, 2001. – 2001. – 600с.
6. Прата С. Язык программирования С. Лекции и упражнения, 5-е издание.: Пер. с англ. – М.: Издательский дом “Вильямс”, 2006. – 960с.
7. Шилдт Г. Полный справочник по С. 4-е издание.: Пер. с англ. – М.: Издательский дом “Вильямс”, 2002. – 704с.
8. Харбисон С., Стил Г. Язык программирования С.: Пер. с англ. – М: ООО Бином Пресс, 2004. – 528с.