ФГОБУ ВПО "СибГУТИ"
Кафедра вычислительных систем

Дисциплины
"ЯЗЫКИ ПРОГРАММИРОВАНИЯ"
"ПРОГРАММИРОВАНИЕ"

Разработка простейших программ на языке Си

Преподаватель:

Доцент Кафедры ВС, к.т.н.

Поляков Артем Юрьевич

Язык Си

Си (англ. C) – язык программирования, разработанный в 1969 – 1973 годах сотрудником AT&T Bell Labs Деннисом Ритчи (Dennis Ritchie).

Реализует *императивную парадигму* программирования: программа представляет собой набор директив предназначенных для исполнителя.

Ориентирован на *структурное программирование*. Данная парадигма предполагает следующее:

1) программа состоит из трех базовых конструкций: *следование*, *ветвление*, *цикл*;

2) базовые конструкции могут быть вложены друг в друга;

3) логически законченные фрагменты программы оформляются в виде подпрограмм.

Язык Си (2)

Конструкции языка эффективно отображаются на машинные инструкции. Поэтому Си часто называют *низкоуровневым* языком. Благодаря этим свойствам язык Си интенсивно применяется при разработке системного программного обеспечения:

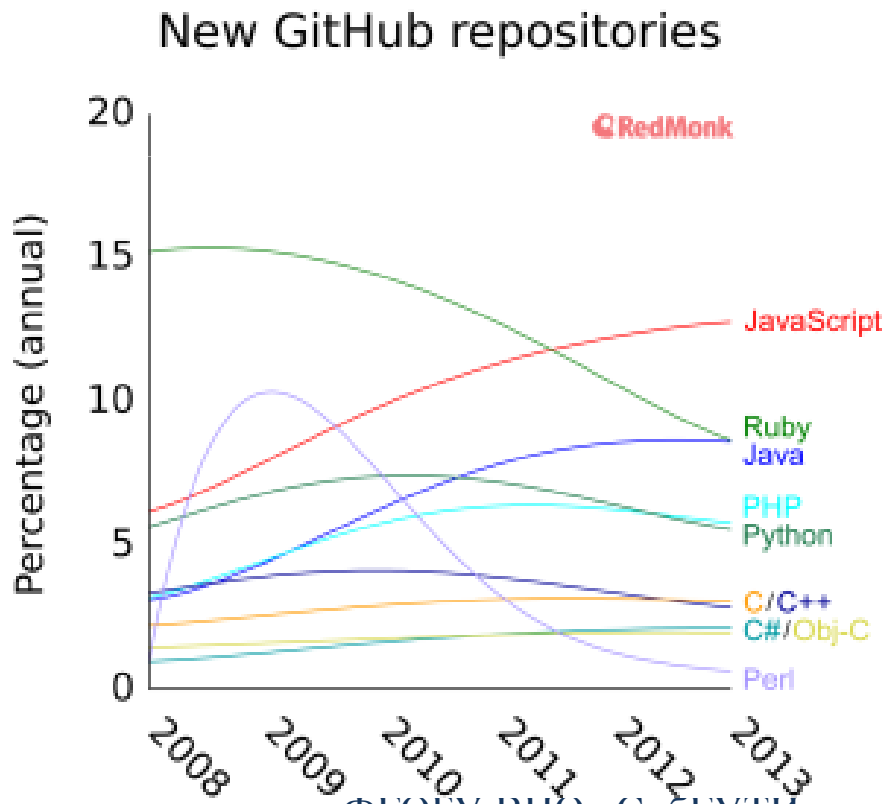
- ОС UNIX
- Linux kernel
- GNU tools (GNU – GNU is Not UNIX — «GNU не UNIX»)
- Open MPI / MPICH2
- 7Zip
- Orwell Dev-C++
- ganglia

Си оказал существенное влияние на развитие индустрии программного обеспечения, а его синтаксис стал основой для таких языков программирования как C++, C#, Java и D.

Популярность языка Си

Для оценки популярности языка программирования используют различные критерии:

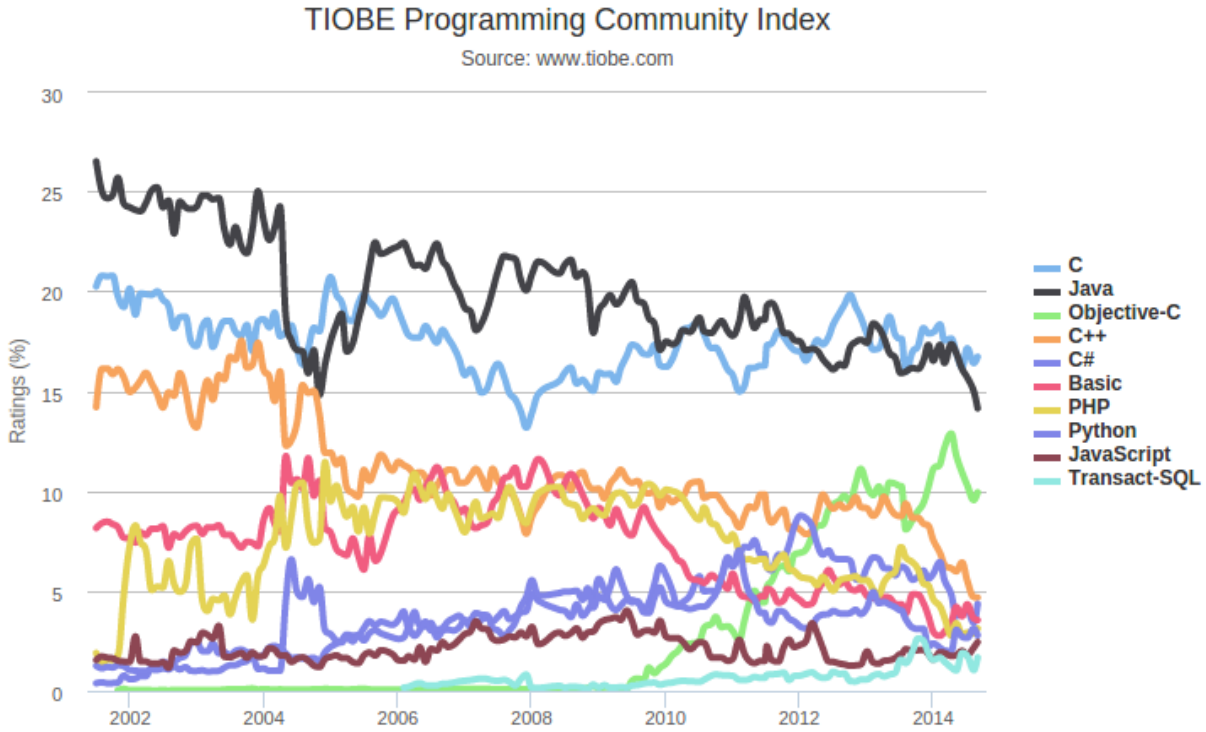
1. Количество и/или качество написанного кода.
2. **Количество новых проектов, использующих данный язык**.
3. Анализ поисковых запросов в популярных поисковиках.



Популярность языка Си (2)

Для оценки популярности языка программирования используют различные критерии:

1. Количество и/или качество написанного кода.
2. Количество новых проектов, использующих данный язык.
3. **Анализ поисковых запросов в популярных поисковиках**.



Каркас простейшей программы [Си/Linux]

```
// C program template
// Символы "//" используются для комментариев

#include <stdio.h> // <-- Функции ввода-вывода

int main()         // <-- Главная подпрограмма
{
    // Код программы
    return 0;      // <-- Код завершения программы
}
```

- Программа на языке Си представляет собой один или несколько *текстовых файлов* с расширением **".c"**.
- Дополнительно могут использоваться *заголовочные файлы* с расширением **".h"** (2 семестр)
- Для преобразования текстового файла в исполняемый файл используется компилятор **gcc**.

Программа на языке Си

Процесс написания программы на языке Си сводится к следующим шагам.

1. **Описание данных**: язык Си требует, чтобы каждая область памяти в программе была явно описана. Описание заключается в указании *имени области* и ее *типа данного*. Тип данного определяет какие данные могут храниться в переменной (целые или вещественные) и какой размер ячейки в байтах (следовательно и диапазон значений).

2. **Преобразование данных**: преобразование данных осуществляется с помощью базовых конструкций языка: *следование*, *ветвление*, *цикл*.

3. **Организация взаимодействия с окружающей средой**:

1) с локальной операционной системой для *настройки параметров исполнения* и *ввода-вывода*;

2) другими процессами на локальной или удаленной машине для осуществления *ввода-вывода*.

ОПИСАНИЕ ДАННЫХ

Описание данных

Области памяти, доступные в языке Си, можно классифицировать следующим образом.

1. **Переменные базовых типов данных.**
2. Массивы базовых типов данных.
3. Переменные и массивы сложных типов данных.
4. Неименованные области (динамическая память).

На первом этапе освоения языка будем использовать только переменные базовых типов.

Описание такой переменной имеет следующий формат:

```
<тип> <имя1> [=<знач>] {, <имя2> [=<знач>] };
```

Например:

```
int i, test, flag;
float var, discriminant;
```

При описании синтаксиса языка будем использовать следующие обозначения:
{ ***x*** } – *x* повторяется 0 и более раз:
int *x* {, *x*}; => int *x*; или int *x*, *x*, *x*, *x*;
[x] – x повторяется 0 или 1 раз:
int *x* [, *x*]; => int *x*; или int *x*, *x*;

Характеристики переменных

Переменная – область памяти, предназначенная для хранения данных.

В языках высокого уровня переменная характеризуется:

- **именем** (идентификатором), которое позволяет компилятору отличать данную ячейку от множества других, используемых в программе;

- **типом данного**, который определяет какую информацию можно хранить в данной ячейке (целые или вещественные числа). Тип данного определяет:

1) формат представления информации;

2) диапазон хранимых значений (размер ячейки памяти);

3) набором допустимых операций.

Идентификация программных элементов [Си]

Компьютерная программа средней сложности содержит тысячи программных объектов. Для того, чтобы уникально идентифицировать их используются **"идентификаторы".**

Идентификатор представляет собой последовательность строчных ("**a – z**") и прописных ("**A – Z**") букв латинского алфавита, а также цифр ("**0 – 9**") и знака подчеркивания ("_").
Идентификатор **начинается** либо **с буквы**, либо **со знака нижнего подчеркивания**.

Идентификаторы используются для:
1) формирования имен объектов языка: *переменных* (ячеек памяти для хранения данных) *и функций* (групп инструкций программы).
2) ключевых слов языка: *типы данных* (**int**, **char**, **float**), *управляющие конструкции* (**for**, **while**, **if**).

Примеры идентификаторов [Си]

Идентификатор – последовательность строчных ("**a – z**") и прописных ("**A – Z**") букв латинского алфавита, а также цифр ("**0 – 9**") и знака подчеркивания ("_").
Идентификатор **начинается** либо **с буквы**, либо **со знака нижнего подчеркивания**.

Допустимые	**Недопустимые**
i	**123**
int	**_@**
test1234	**test~**
Q1W2E3R4	**1Q1W2E3R4**
S123456789	**-S123456789**
_123456789	**_1234=56789**
John_Deer	**John␣Deer**

Характеристики переменных (2)

Переменная – область памяти, предназначенная для хранения данных.

В языках высокого уровня переменная характеризуется:

• **именем** (идентификатором), которое позволяет компилятору отличать данную ячейку от множества других, используемых в программе;

• **типом данного**, который определяет какую информацию можно хранить в данной ячейке (целые или вещественные числа). Тип данного определяет:

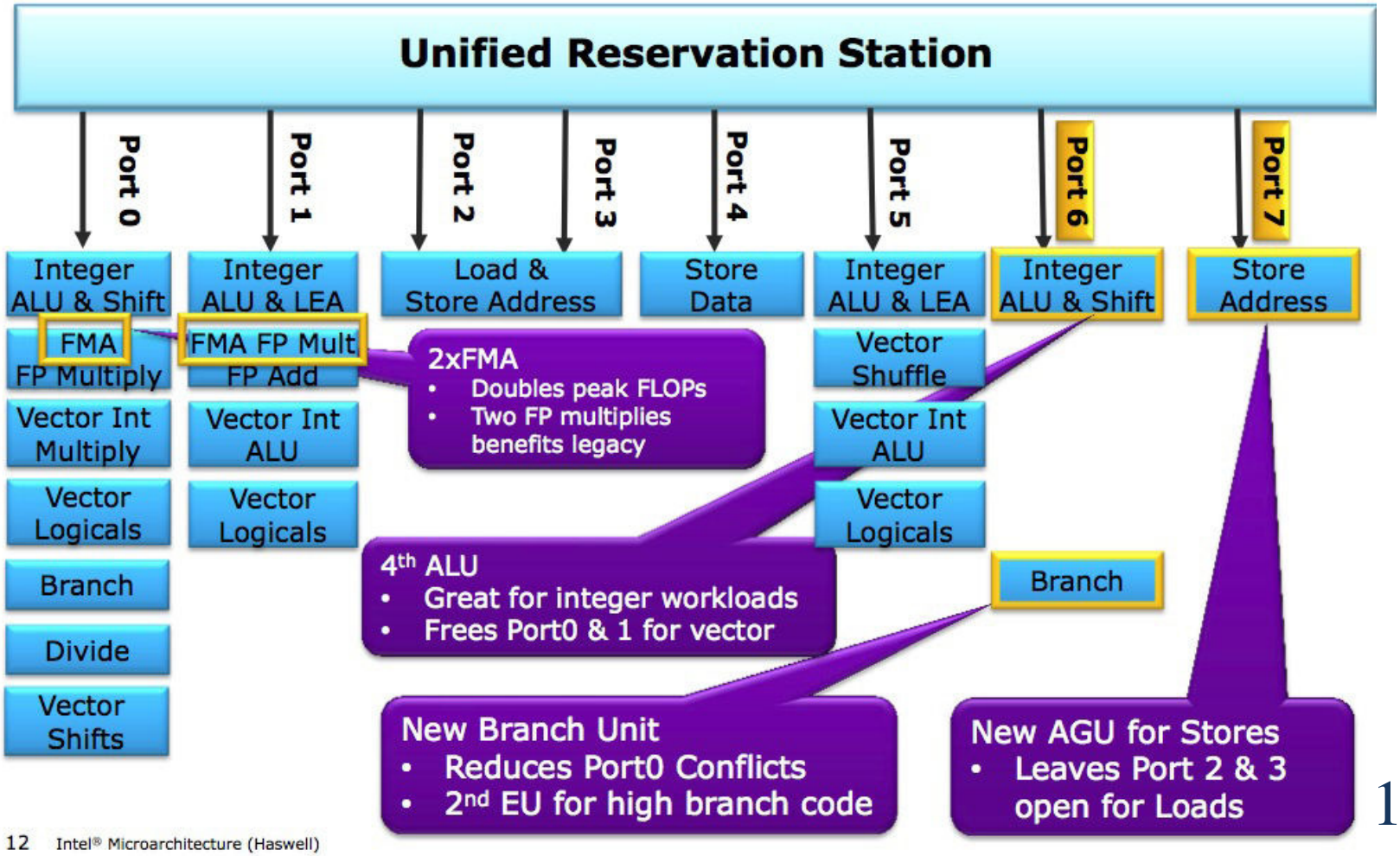
1) формат представления информации;

2) диапазон хранимых значений (размер ячейки памяти);

3) набором допустимых операций.

Разделение целочисленных и вещественных операций

Haswell Execution Unit Overview



Некоторые команды процессора (архитектура x86)

ЦЕЛОЧИСЛЕННЫЕ

Арифметические

ADD – сложение
SUB – вычитание
MUL – умножение
DIV – деление нацело
IMUL – *знаковое* умножение
IDIV – *знаковое* деление

Логические

AND – логическое И
OR – логическое ИЛИ
XOR – исключающее ИЛИ

Сдвиги

SHL/SHR – сдвиг влево/вправо
SAL/SAR – арифметический (*знаковый*) сдвиг влево/вправо

Над числами с ПЛАВАЮЩЕЙ ТОЧКОЙ

Арифметические

FADD – сложение
FSUB – вычитание
FMUL – умножение
FDIV – деление
FIMUL – умножить вещественное число на целое
FIDIV – разделить вещественное число на целое
FSIN – вычислить синус
FSINCOS – вычислить синус и косинус

Тип данного

<тип> <имя1> {, <имя2>};

Тип данного определяет какие данные могут храниться в переменной. В вычислительной технике различают *целые* и *вещественные* данные. Они имеют *различное внутреннее представление* и обрабатываются *различными исполняемыми элементами процессора* (CPU execution units).

Целые типы используют для хранения:

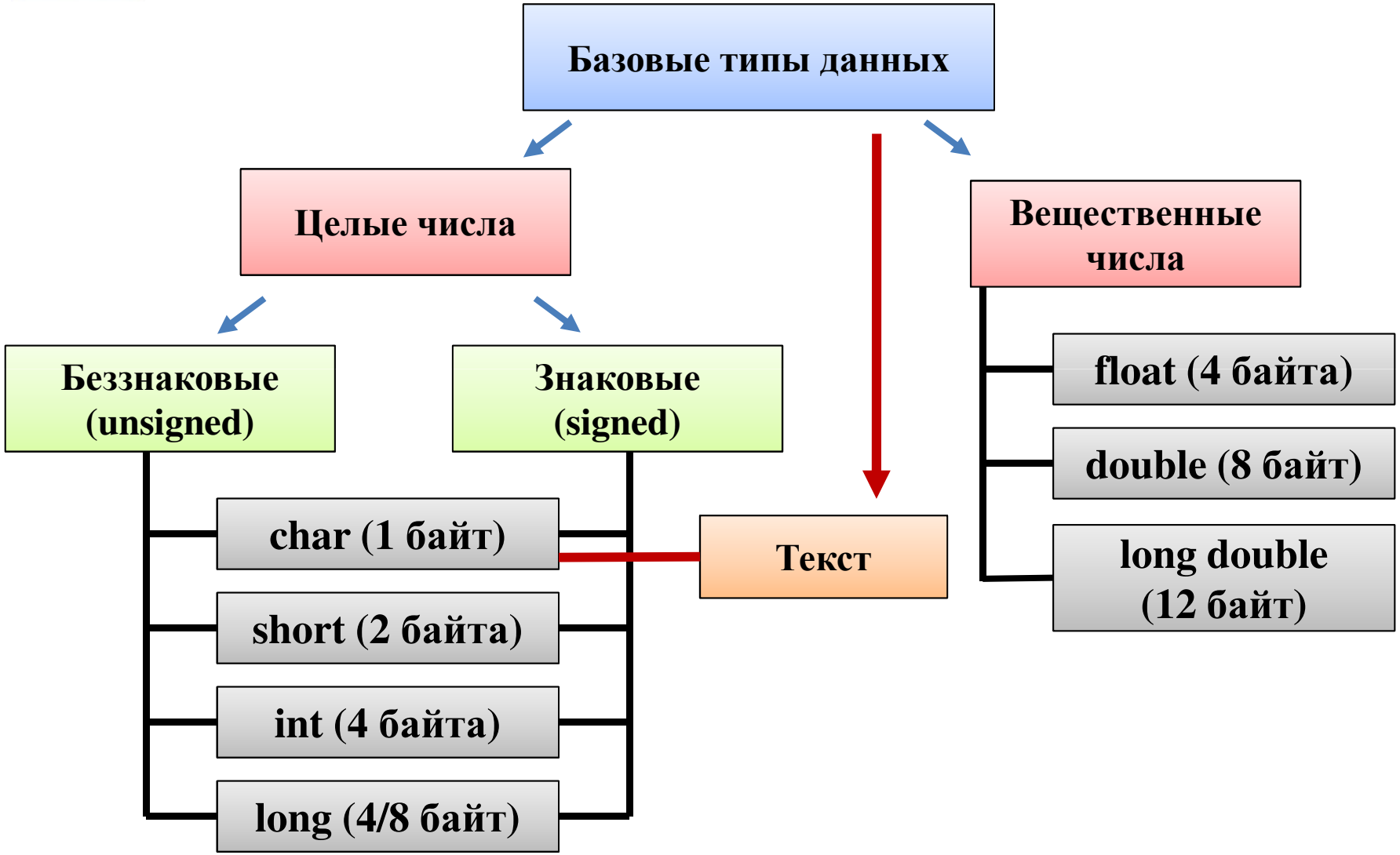
1) целых чисел (со знаком и без знака): типы **char, short, int, long int, long long int**;

2) символьных данных (текста, представленного последовательностью ASCII-кодов), для этого обычно применяются однобайтовый целый тип **char**.

Вещественные типы используются для хранения действительных чисел. Доступные типы данных: **float, double, long double**;

Как *целочисленные*, так и *вещественные* типы данных различаются *диапазоном значений*, который зависит от *размера выделяемой памяти*.

Базовые типы данных [Си]



Целые типы данных [Си]

Беззнаковые целые

Обозначение	Размер, байт	Диапазон значений
unsigned char	1	[0; 255]
unsigned short	2	[0; 65535]
unsigned int	4	[0; 4294967295]
unsigned long	4 или 8	[0; 2^{64} - 1] (8 байт)

Знаковые целые

Обозначение	Размер, байт	Диапазон значений
[signed] char	1	[-128; 127]
[signed] short	2	[-32768; 32767]
[signed] int	4	[–2147483648; 2147483647]
[signed] long	4 или 8	[-2^{63}; 2^{63} - 1] (8 байт)

[x] – x повторяется 0 или 1 раз: [signed] **int *x*; => int *x*;** или **signed int *x*;**

Вещественные типы данных [Си]

Идентификатор	Размер, байт	Диапазон значений
float	4	от $\pm 3.4 \cdot 10^{-38}$ до $\pm 3.4 \cdot 10^{38}$ (~ 7 значащих цифр)
double	8	от $\pm 1.7 \cdot 10^{-308}$ до $\pm 1.7 \cdot 10^{308}$ (~ 15 значащих цифр)
long double	12	от $\pm 1.2 \cdot 10^{-4932}$ до $\pm 1.2 \cdot 10^{4932}$ (~22 значащие цифры)

Вещественные типы данных [Си] (2)

Идентификатор	Размер, байт	Диапазон значений
float	4	от $\pm 3.4 \cdot 10^{-38}$ до $\pm 3.4 \cdot 10^{38}$ (~ 7 значащих цифр)
double	8	от $\pm 1.7 \cdot 10^{-308}$ до $\pm 1.7 \cdot 10^{308}$ (~ 15 значащих цифр)
long double	12	от $\pm 1.2 \cdot 10^{-4932}$ до $\pm 1.2 \cdot 10^{4932}$ (~22 значащие цифры)

Диапазон значений 8-байтового целого типа (unsigned long) на 9 порядков меньше, чем у 4-хбайтового вещественного типа (float).

Обозначение	Размер, байт	Диапазон значений
unsigned long	8	$2^{64} - 1 \sim 1.84 \cdot 10^{19}$ (~19 значащих цифр)

Объявление переменных

```
#include <stdio.h>

int main()
{
    int I = 0, IntegerVariable;
    unsigned short short_val;
    char c = 'A';
    float f, Fl = 1.2, f1, f2, f3;
    double f458 = 1E-5;
    signed int sint;

    . . .

}
```

Выбор типа данного

Тип	Беззнаковый	Знаковый
`char`	`[0; 255]`	`[-128; 127]`
`short`	`[0; 65535]`	`[-32768; 32767]`
`int`	`[0; 4294967295]`	`[-2147483648; 2147483647]`

6. Государственный долг США
(Табло в Нью-Йорке около Таймс-сквер)

Компиляция типов данных

int.c

```
#include <stdio.h>

int main()
{
    int a = 5, b = 6, c;
    c = a + b;
}
```

float.c

```
#include <stdio.h>

int main()
{
    float a = 5, b = 6, c;
    c = a + b;
}
```

gcc -S int.c

int.s

```
    ...
    movl   -8(%rbp), %eax
    movl   -12(%rbp), %edx
    addl   %edx, %eax
    ...
```

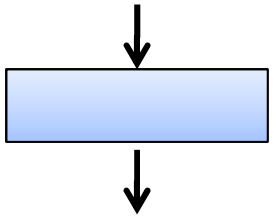
gcc -S -mno-sse float.c

float.s

```
    ...
    flds    -12(%rbp)
    fadds   -8(%rbp)
    fstps   -4(%rbp)
    ...
```

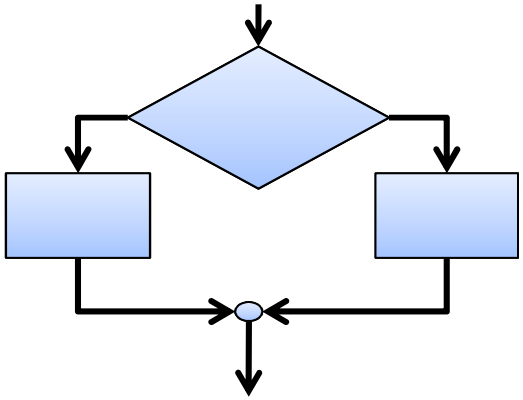
ПРЕОБРАЗОВАНИЕ ДАННЫХ

Базовые конструкции

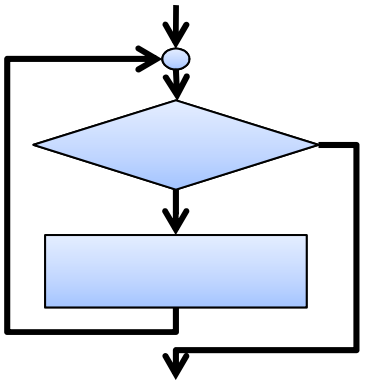


Следование

Ветвление

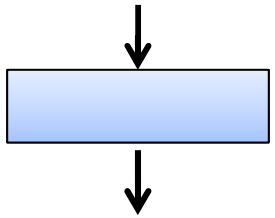


Цикл



Конструкция следование

Следование



Операторы в языке Си выполняются *последовательно*. После завершения текущего оператора производится переход к следующему оператору по тексту программы.

Исключением являются: операторы, входящие в состав ветвления или цикла.

Например:

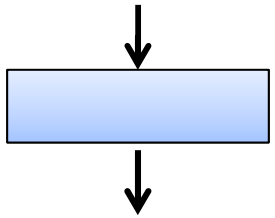
```
// Определение переменных
int a = 10, b = 5, c;
// операторы
c = a + b;
c = c * 2;
a = c / b;
b = a * 1.5;
```

Блок операторов

Определения переменных и набор операторов могут быть сгруппированы вместе при помощи фигурных скобок.

Для всех элементов программы, расположенных вне этих скобок все операторы внутри видны как один составной оператор:

Следование

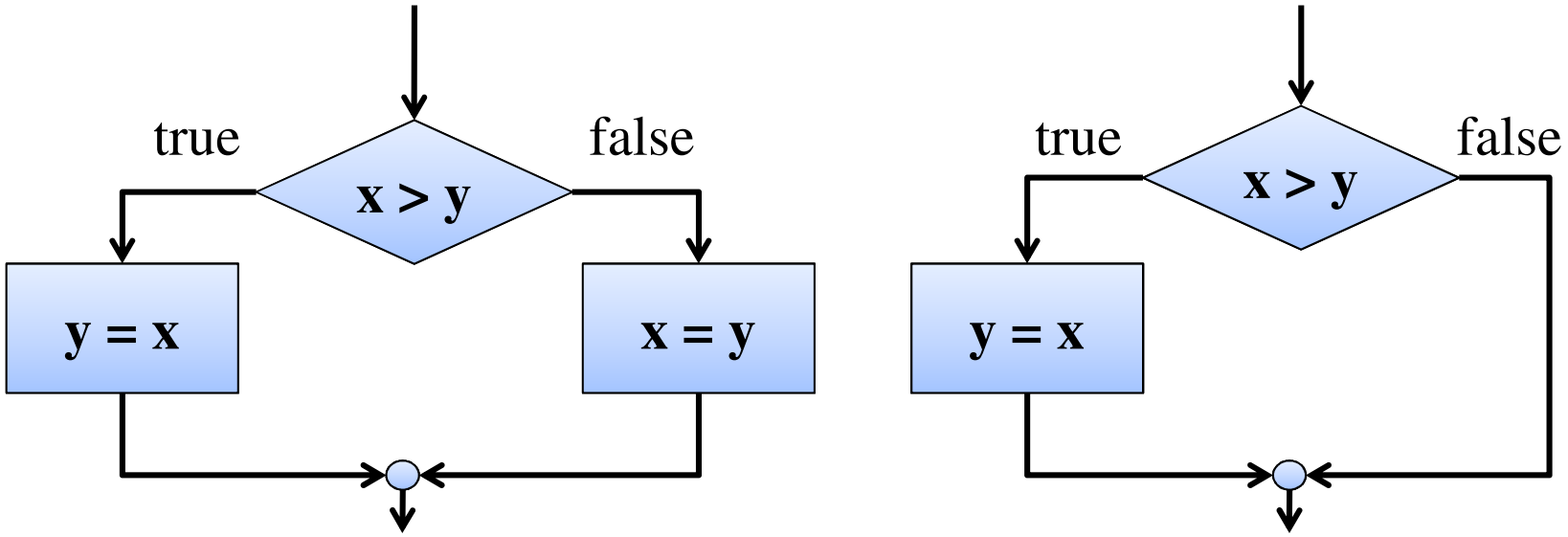


```
// Определение переменных
int a = 10, b = 5, c;
// составной оператор
{
    int t;
    t = a;
    a = b;
    b = t;
}
c = a + b;
c = c * 2;
```

Ветвление

Ветвление – управляющая структура, организующая выполнение лишь одного из двух указанных действий в зависимости от справедливости некоторого условия.

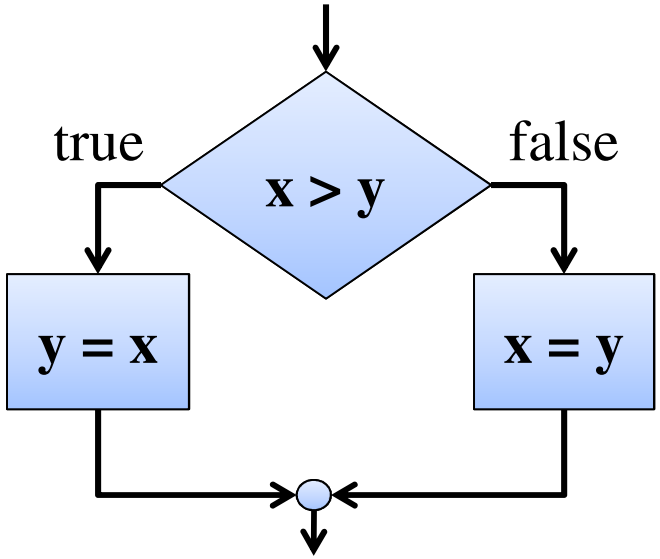
Условие – высказывание, которое может быть истинно (true) или ложно (false). Запись ветвления выполняется в двух формах: полной (слева) и неполной (справа).



Оператор ветвления [Си]

```
if( <Выражение> )
    <Оператор1>
[ else
    <Оператор2> ]
```

```
if( x > y )
    y = x;
else
    x = y;
```

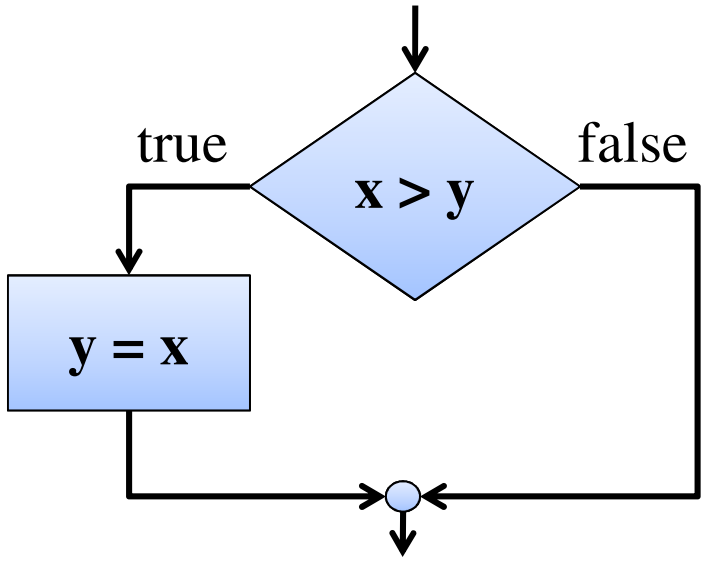


Оператор ветвления [Си]

```
if( <Выражение> )
    <Оператор1>
[ else
    <Оператор2> ]
```

Части синтаксических конструкций, не обязательные для использования, указываются в квадратных случаях.
В данном случае неполная версия ветвления не требует ветви "else".

```
if( x > y )
    y = x;
```



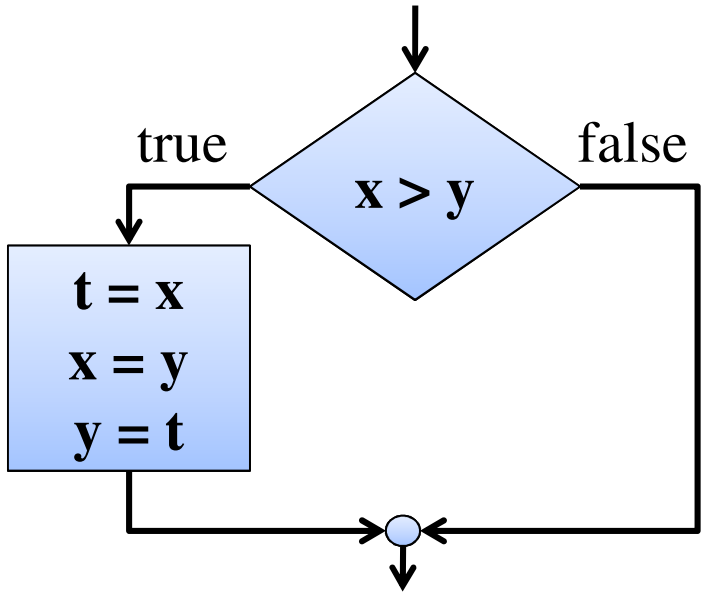
Составной оператор [Си]

Ветвление в языке СИ предусматривают наличие *только одного оператора*.

Для того, чтобы выполнить несколько операторов в рамках некоторой ветви необходимо заключить их в *фигурные скобки*. В результате оператор ветвления будет работать с одним сложным (составным) оператором, содержащим группу более простых.

Табуляция

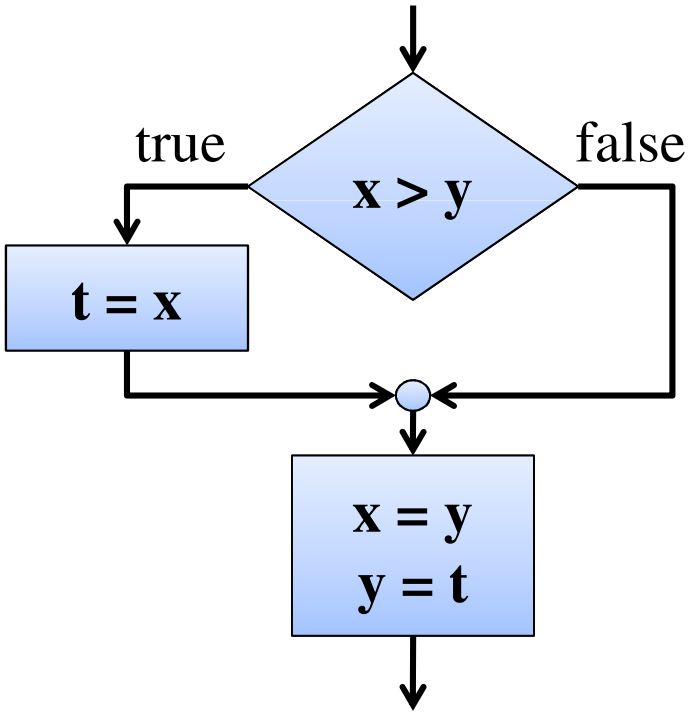
```
        . . .
if( x > y ){
  → int t;
  → t = x;
  → x = y;
  → y = t;
}
        . . .
```



Составной оператор [Си] (2)

Отсутствие фигурных скобок приведет к неправильной интерпретации алгоритма компилятором. Например:

```
        . . .
if( x > y )
    t = x;
x = y;
y = t;
        . . .
```



Вычисление корней квадратного уравнения

$$a \cdot x^2 + b \cdot x + c = 0$$
$$D = b^2 - 4 \cdot a \cdot c$$

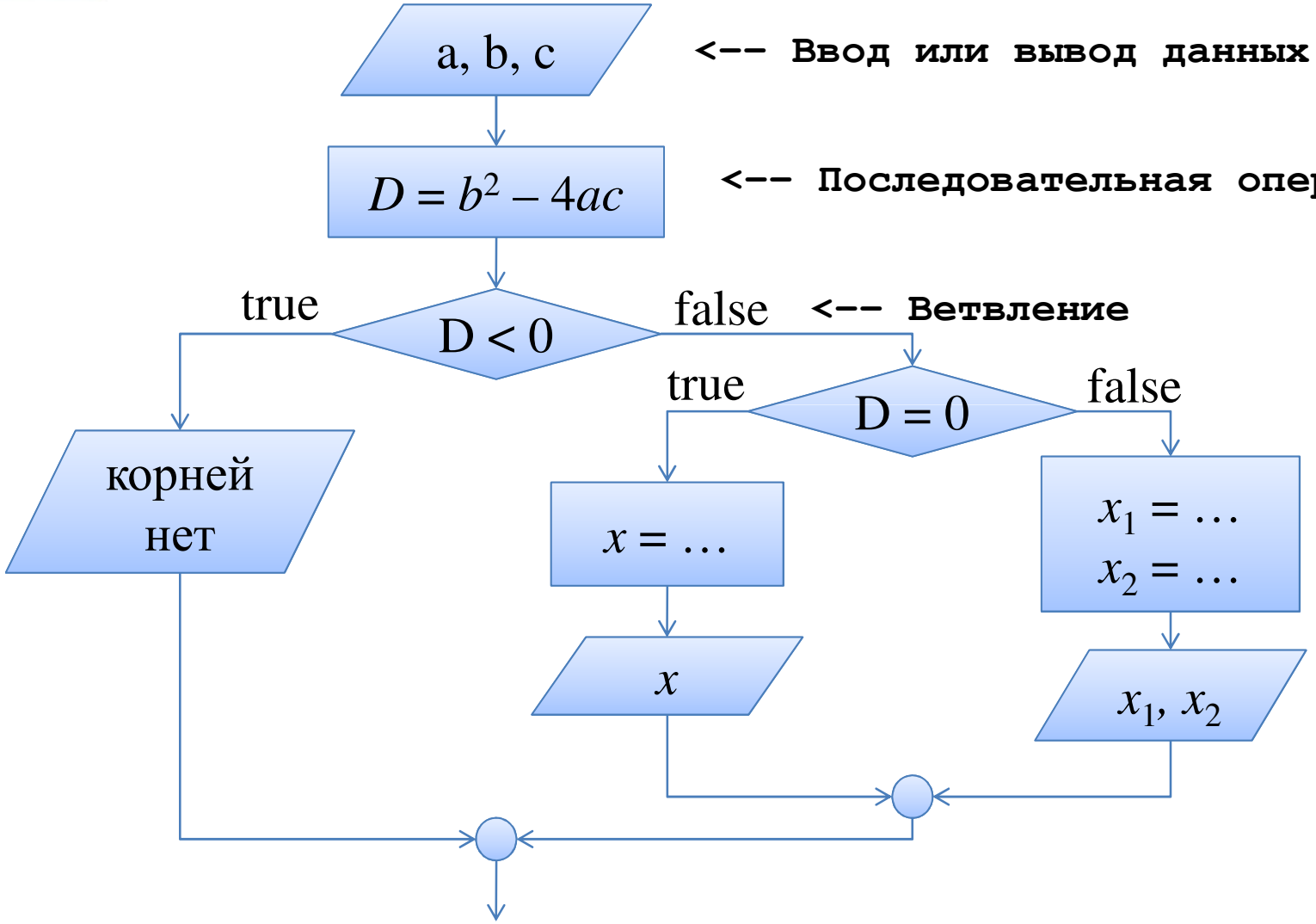
1) Если D < 0 – **корней нет**
2) Если D = 0 – **корень один** и вычисляется по формуле:

$$x = \frac{-b}{2a}$$

3) Если D > 0 – корней два, они вычисляются по формулам:

$$x_1 = \frac{-b - \sqrt{D}}{2a}; x_2 = \frac{-b + \sqrt{D}}{2a}$$

Алгоритм решения квадратного уравнения

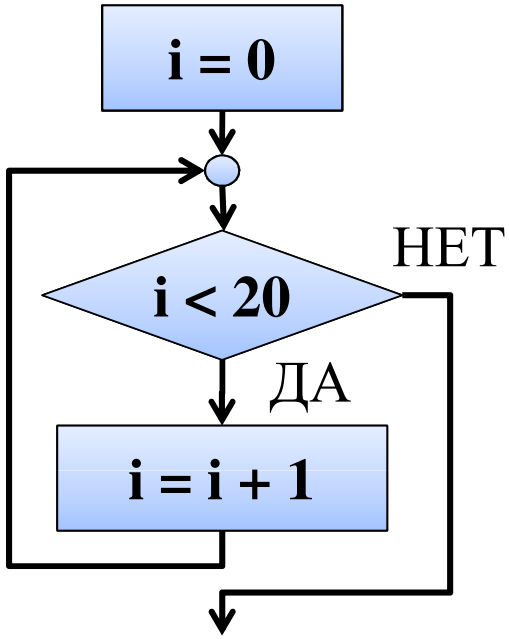


Программа вычисление корней квадратного уравнения

```
#include <stdio.h>
int main()
{
	float a, b, c, x1, x2, D;
	scanf("%f %f %f", &a, &b, &c);
	D = b*b - 4*a*c;
	if( D < 0 )
		printf("No roots\n");
	else if( D == 0 ){
		x1 =  -b/(2*a);
		printf("One root: %f\n", x1);
	} else {
		x1 = (-b - sqrt(D)) / (2*a);
		x2 = (-b + sqrt(D)) / (2*a);
		printf("Two roots: %f, %f\n", x1, x2);
	}
	return 0;
}
```

Цикл

Цикл – разновидность управляющей конструкции в высокоуровневых языках программирования. Предназначена для организации *многократного* исполнения *однотипных инструкций* над *различными данными*.

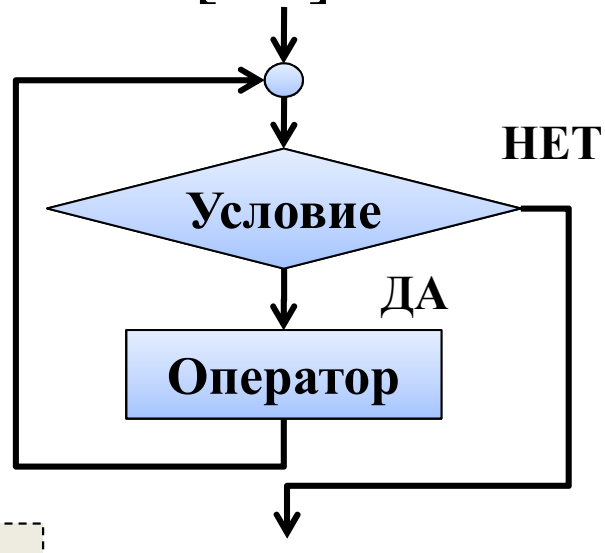


В языке Си предусмотрено 3 циклических конструкции, которые являются *взаимозаменяемыми*:

1) с предусловием – **while**, **for**;
2) с постусловием – **do-while**.

Циклический оператор **while** [Си]

```
while( <Условие> )
    <Оператор>
```

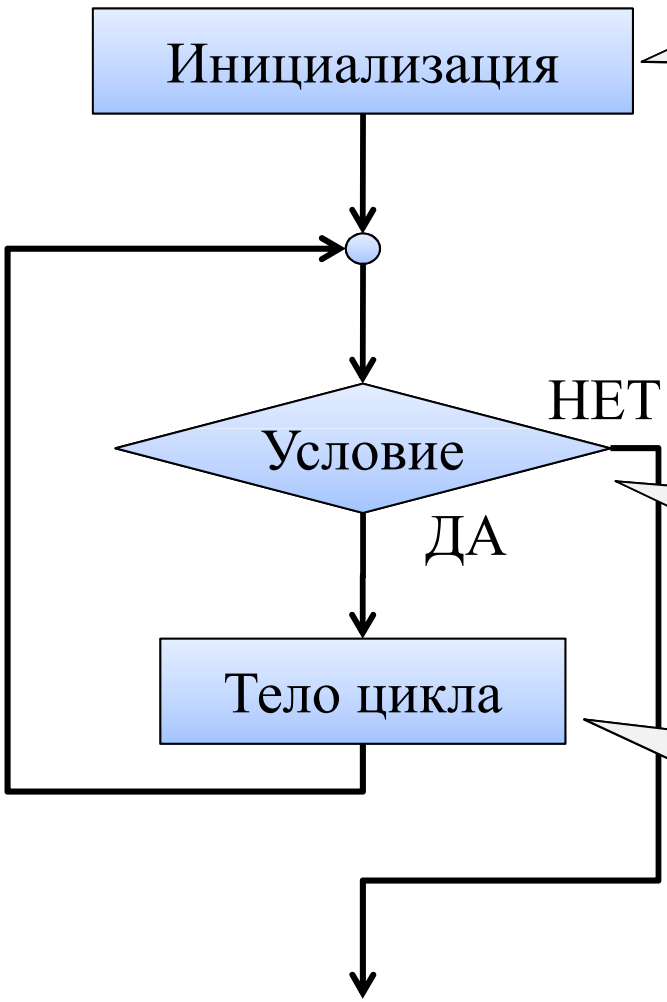



```
#include <stdio.h>
int main()
{
  int i = 0;
  while( i < 20 )
    i++;

  return 0;
}
```

Перебор целых чисел от 0 до 19

Цикл: основные понятия



Предварительная инициализация переменных, в частности, установка переменной-счетчика.

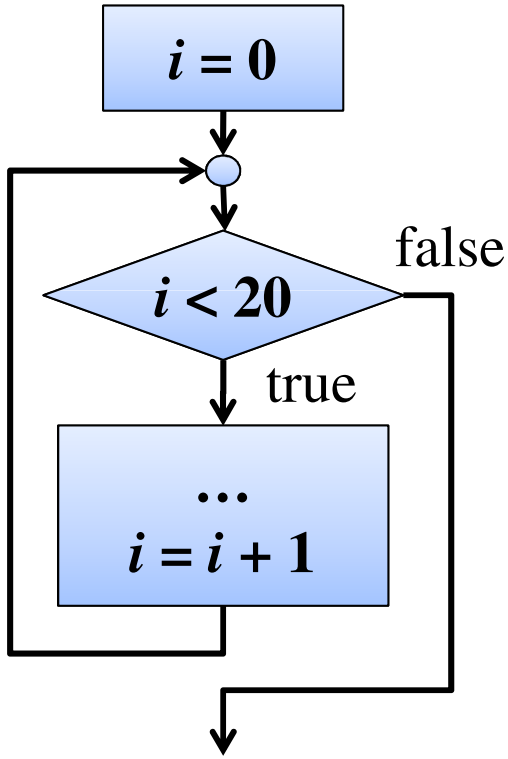
Условие продолжение цикла. Тело цикла выполняется до тех пор, пока условие истинно.

Тело цикла, содержащее "полезные" инструкции, которые необходимо выполнить несколько раз.

Цикл: основные понятия (2)

Итерация цикла – однократное выполнение тела цикла.

Счетчик цикла – термин, используемый для обозначения переменной, контролирующей повторы выполнения циклов. Счетчики циклов изменяют свое значение при каждом прохождении цикла, подставляя уникальное значение для каждой отдельной итерации.



Счетчик цикла также используется для определения момента, когда цикл должен **завершить свою работу**, после чего программа продолжает свое выполнение, обратившись к инструкции, расположенной после цикла.

Обычно счетчики называют *i*, *j* и *k*.

Применение циклов

Циклические конструкции позволяют автоматизировать объемные вычисления, требующие многократного повторения однотипных действий над различными данными.

Для того, чтобы записать цикл, позволяющий решить поставленную задачу необходимо:

1) выделить повторяющиеся операции и определить набор *S* переменных, которые должны изменяться на каждой итерации цикла, и правила по которым они изменяются;

2) определить условие продолжения цикла, истинность которого говорит о том, что необходимо выполнить тело цикла как минимум еще один раз;

3) выписать начальные значения переменных в наборе *S*;

4) оформить цикл в соответствии с синтаксисом языка Си.

Вычисление суммы элементов числовой последовательности

Задача:

На вход программы поступает количество N чисел в последовательности, а потом элементы a_1, a_2, … a_N этой последовательности. Необходимо вычислить их сумму.

Решение:

$$S = a_1 + a_2 + \ldots + a_{N-1} + a_N$$

$$S = (\ \ldots(a_1 + a_2) + \ldots + a_{N-1}) + a_N$$

0)	$S = 0$
1)	$S = S + a_1$
2)	$S = S + a_2$
	…
i)	$S = S + a_i$
	…
$N-1$)	$S = S + a_{N-1}$
N)	$S = S + a_N$

Вычисление суммы элементов числовой последовательности

0)	$S = 0$
1)	$S = S + a_1$
2)	$S = S + a_2$
	…
i)	$S = S + a_i$
	…
$N-1$)	$S = S + a_{N-1}$
N)	$S = S + a_N$

1. Решение задачи можно разбить на однотипные фрагменты: прибавление к имеющемуся фрагменту суммы очередного элемента. При этом на каждой итерации цикла вводится элемент a_i, обновляется сумма S ($S = S + a_i$), вычисляется номер i (i=i+1) следующего элемента.

2. Цикл продолжается до тех пор, пока не будет введено N элементов. i – счетчик цикла, определяет количество выполненных итераций. Поэтому условие продолжения цикла можно записать следующим образом: $i \le N$.

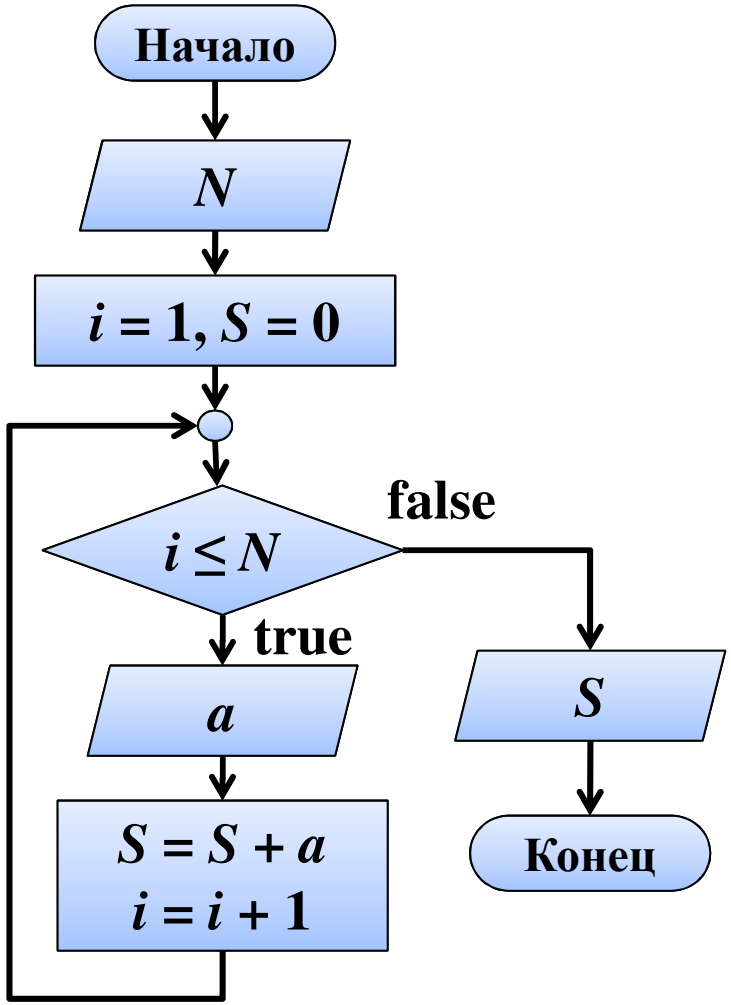
3. Изначально переменные, описывающие состояние цикла нужно установить соответственно:

$i = 1$ – ввод первого элемента.

$S = 0$ – до начала ввода сумма полагается нулевой.

Вычисление суммы элементов числовой последовательности (2)

0)	$S = 0$
1)	$S = S + a_1$
2)	$S = S + a_2$
	…
i)	$S = S + a_i$
	…
$N-1$)	$S = S + a_{N-1}$
N)	$S = S + a_N$



Вложенные циклы

	1	2	3	4	5
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12	13	14	15
4	16	17	18	19	20

При решении многих задач возникает необходимость одновременного перебора (однократного просмотра) значений по нескольким независимым направлениям. Например, перебор координат двумерной области:

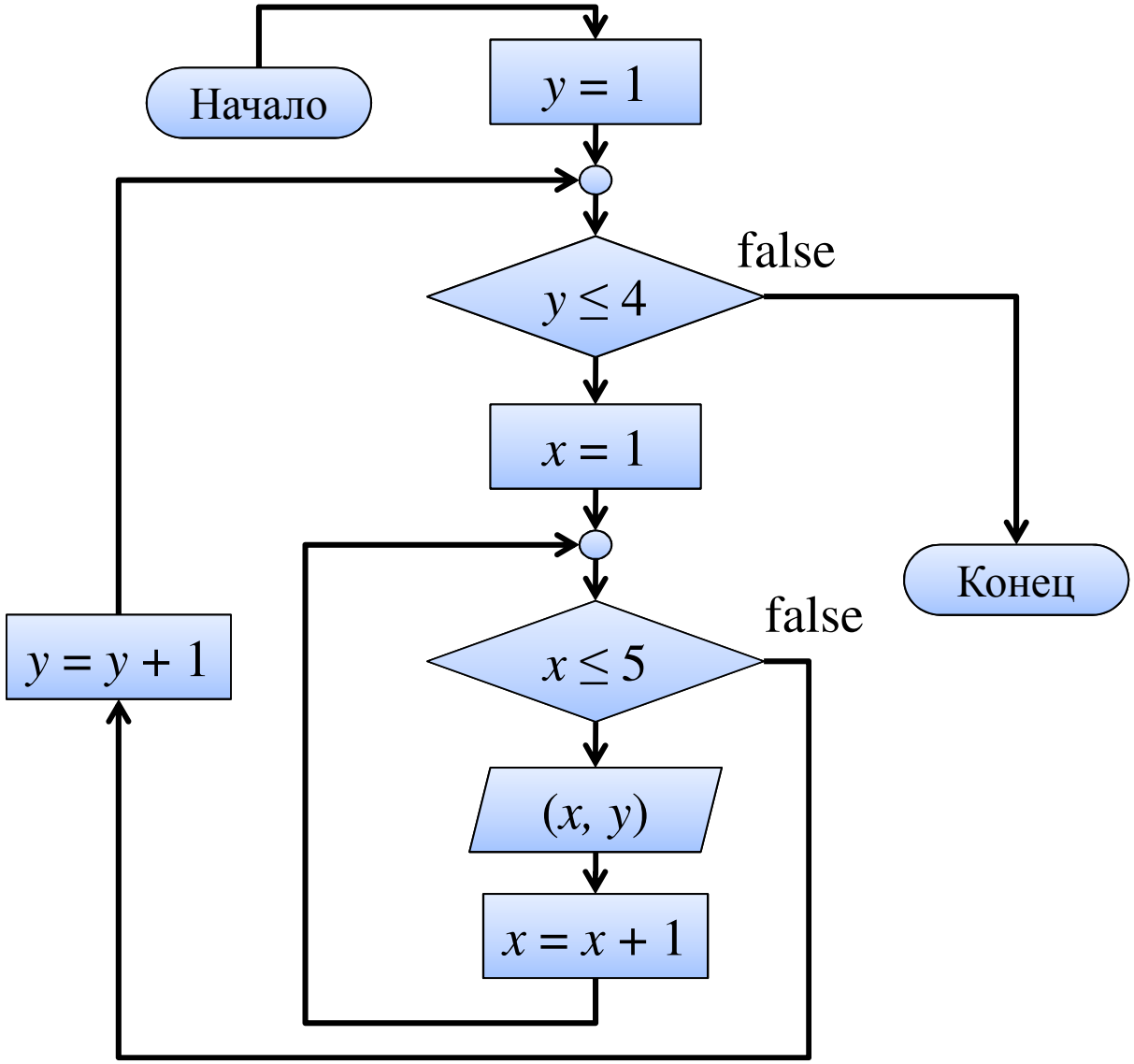
(1,1), (2, 1), (3,1), (4,1), (1,2), (2,2), … , (1,5), …, (4,5)

Для решения этой задачи можно предложить два варианта:

1) организовать один цикл со счетчиком и на каждой итерации выражать координаты текущей области из текущего значения счетчика.

2) организовать два цикла так, чтобы один из них (внутренний) составлял в итерацию другого (внешнего). Счетчик внешнего цикла будет отвечать за координату *y*, а внутреннего – за координату *x*.

Вложенные циклы (2)



Вложенные циклы (3)

```
#include <stdio.h>

int main()
{
  int x, y;

  y = 1;
  while( y <= 4 ){
    x = 1;
    while( x <= 5 ){
      printf("(%d,%d)\n", x, y);
      x++;
    }
    y++;
  }
  return 0;
}
```

	1	2	3	4	5
1	1	2	3	4	5
2	6	7	8	9	10
3	11	12	13	14	15
4	16	17	18	19	20

(1,1) (1,2) (1,3) (1,4)
(2,1) (2,2) (2,3) (2,4)
(3,1) (3,2) (3,3) (3,4)
(4,1) (4,2) (4,3) (4,4)
(5,1) (5,2) (5,3) (5,4)

Вычисление простых чисел в указанном диапазоне

Простое число – это натуральное число, имеющее ровно два различных натуральных делителя: единицу и само себя. Все остальные натуральные числа, кроме единицы, называются составными.

Задача:

Вывести список простых чисел, расположенных в диапазоне от 1 до *N*, где *N* задается с клавиатуры.

Решение:

1. Осуществить перебор всех чисел в диапазоне от 1 до *N* (цикл, в котором счетчик *i* пробегает значения от 1 до *N* с шагом 1).

2. Для каждого числа *i* проверить его простоту. Для этого необходимо проверить, делится ли оно на что-либо кроме 1 и самого себя (цикл, в котором счетчик *j* перебирает все возможные делители *i*, т.е. $j = 2 \ldots i - 1$).

Проверка на простоту

0)	$f = 0$
1)	$f = f$ ИЛИ $(i \,\%\, 2 \neq 0)$
2)	$f = f$ ИЛИ $(i \,\%\, 3 \neq 0)$
	…
j)	$f = f$ ИЛИ $(i \,\%\, j \neq 0)$
	…
$i - 1$)	$f = f$ ИЛИ $(i \,\%\, (i - 1) \neq 0)$

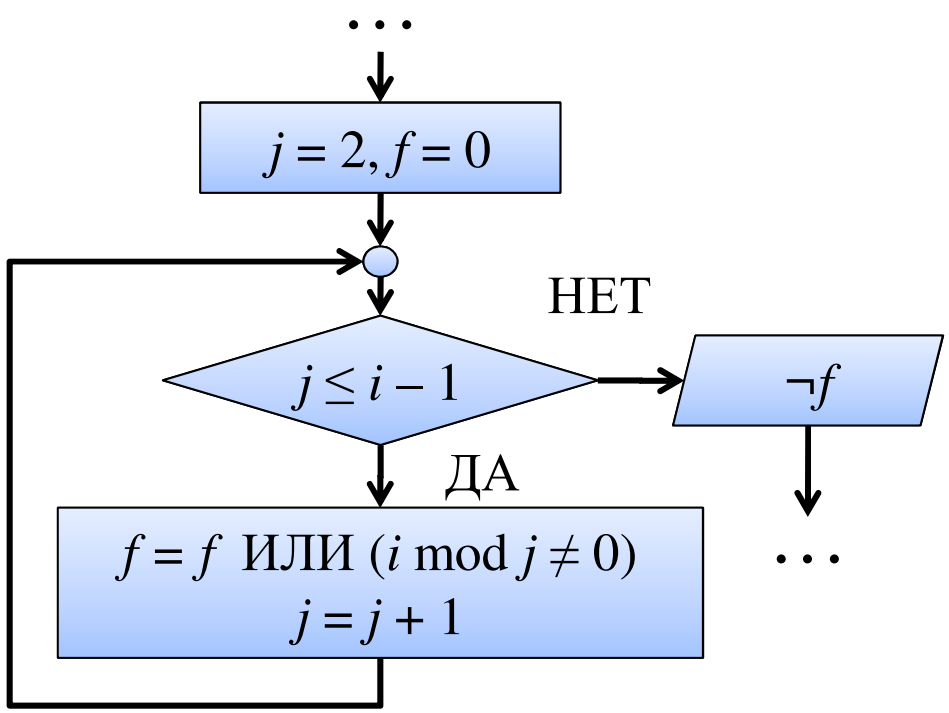
1. Повторяющийся шаг: проверка, делится ли i на текущий потенциальный делитель j. На каждой итерации изменяется j ($j = j + 1$) и флаг f, в котором фиксируется факт деления без остатка. Если по завершению цикла $f = 1$, то число составное.

2. Цикл продолжается до тех пор, пока не будет просмотрено $(i - 1)$ делителей. Условие продолжения цикла: $j < (i - 1)$ можно записать следующим образом: $i \leq N$.

3. Инициализация переменных: $j = 2$ (первый потенциальный делитель), $f = 0$ – изначально предполагаем число простым.

Проверка на простоту

i – целое число, которое проверяется на простоту.
На каждой итерации проверяется потенциальный делитель j, который находится в диапазоне $[2, i-1]$.
Если некоторое значение j является делителем i, то *переменная-флаг* f будет установлена в значение 1.



На выходе значение флага f определяет результат:

- если $f = 0$ ($\neg f = 1$), то среди чисел диапазона $[2, i-1]$ не нашлось ни одного делителя – число простое.
- иначе $f = 1$ ($\neg f = 0$) – число составное.

Переменные-флаги

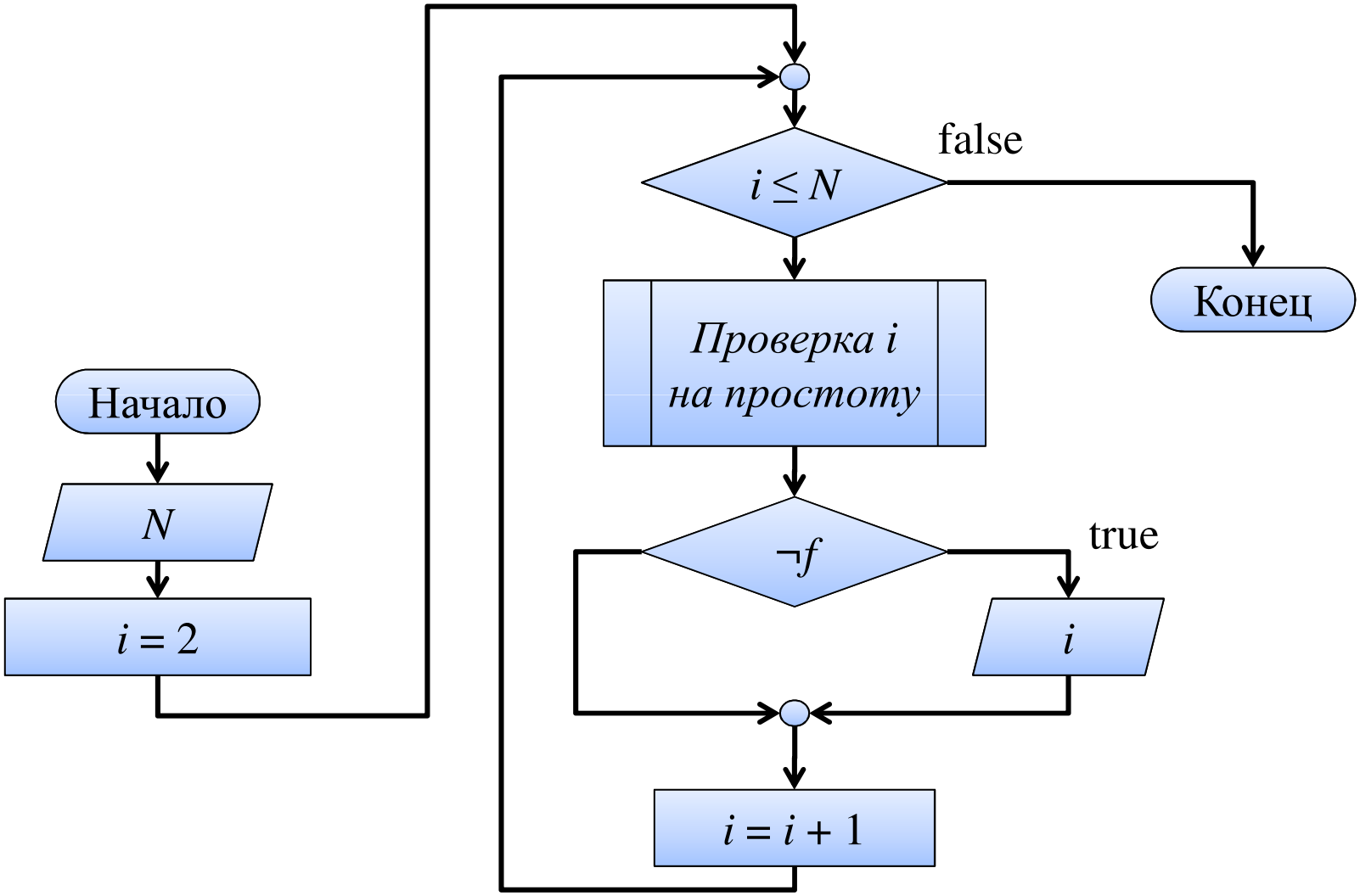
W Флаг (нидерл. vlag) — полотнище правильной геометрической (чаще всего, прямоугольной) формы, имеющее какую-либо специальную расцветку. Обычно флаг поднимается на специальной мачте (флагштоке).

Одним из применений флагов является передача сигналов. В простейшем случае с помощью флага можно передать информацию объемом 1 бит: **флаг поднят** или **флаг не поднят**.

Переменная флаг – это, как правило, переменная логического типа, значение которой сигнализирует о состоянии вычислительного процесса.

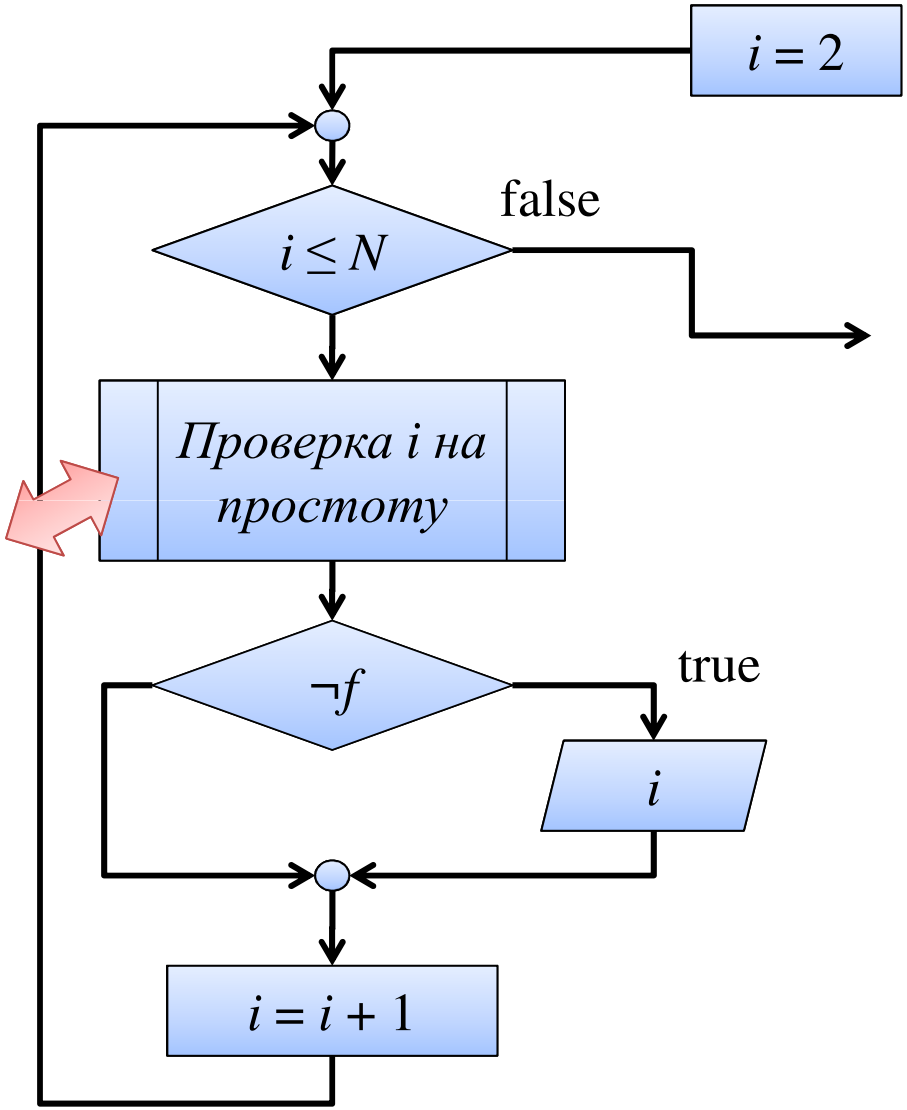
В примере, рассмотренном на предыдущем слайде, переменная-флаг *f* описывает состояние процесса с точки зрения обнаружения делителей числа *i*.

Перебор чисел в заданном диапазоне



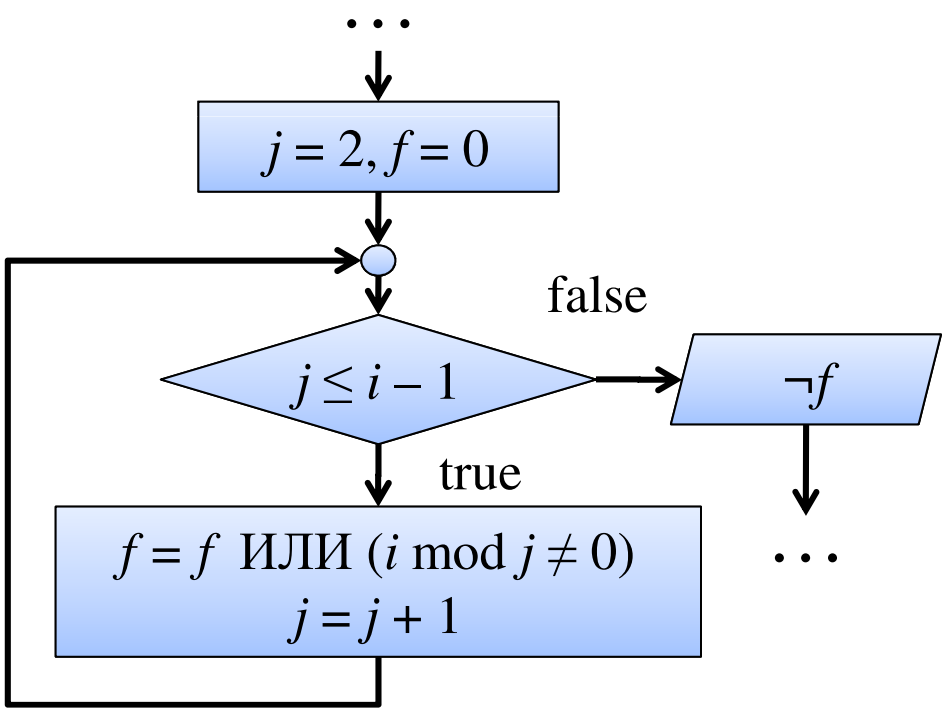
Вычисление простых чисел в указанном диапазоне

```
#include <stdio.h>
int main()
{
  int N, i = 2, j, f;
  printf("Input N: ");
  scanf("%d", &N);
  while( i <= N ){
    f = 0; j = 2;
    while( j <= i - 1 ){
      f = f || ( i%j == 0);
      j++;
    }
    if( !f ){
      printf("%d ", i);
    }
    i++;
  }
}
```



Досрочное завершение цикла

В некоторых случаях возникает необходимость досрочного завершения цикла. Например, при проверке простоты числа, обнаружение первого делителя гарантирует, что число не является простым и дальнейшая проверка не нужна.



Особенностью "досрочного" завершения является то, что заранее неизвестно на какой итерации оно потребуется.

Существует несколько способов организации досрочного завершения:

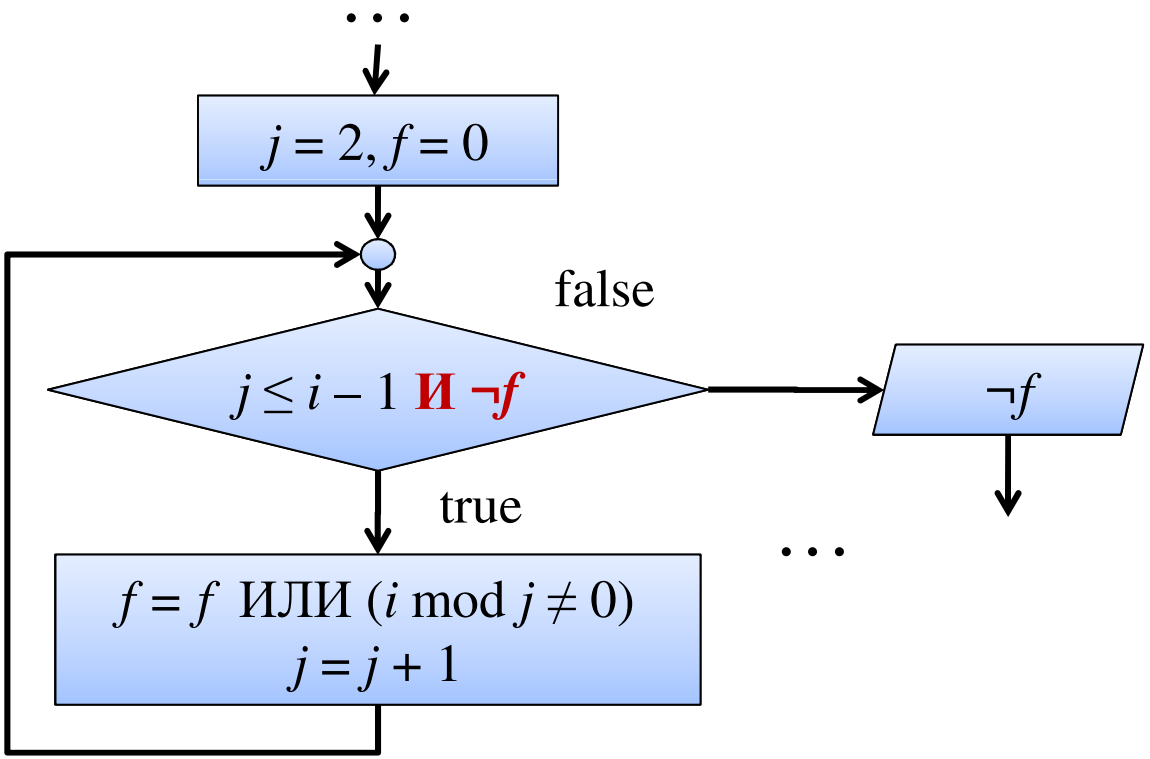
1. Оператор break
2. Оператор goto

безусловный переход

3. **Использование переменной-флага.**

Досрочное завершение цикла

Для организации досрочного завершения цикла с помощью переменной-флага необходимо включить в состав условия продолжения цикла логическое выражение, зависящее от флага и принимающее ложное значение в случае, когда дальнейшее выполнение тела цикла не требуется.



Инструкции перехода (x86)

Безусловный переход

0x1000	instr #1
0x1004	instr #2
0x1008	instr #3
0x100c	intsr #4
0x1010	jmp 0x1024
0x1014	instr #7
0x1018	instr #8
0x101c	instr #9
0x1020	instr #10
0x1024	instr #11
0x1028	instr #12
0x102c	instr #13

Существует несколько команд безусловного перехода.

1. CALL – вызов процедуры.
2. RET – возврат из процедуры.
3. INT – вызов программного прерывания.
4. **JMP – переход (прыжок) к указанному адресу.**

Условный переход

Команда процессору на изменение порядка выполнения программы в соответствии с результатом проверки некоторого условия.

```
0x1000    instr #1
0x1004    instr #2
0x1008    cmp %eax %ebx
0x100c    jle 0x1018
0x1010    instr #3
0x1014    instr #4
0x1018    instr #5
0x101c    instr #6
```

Сравнить значения регистров

Если %eax < %ebx перейти

Команды условного перехода

Обычно имеет две стадии.

1. Сравнение некоторых величин (X и Y), от которых зависит дальнейшее выполнение программы.

```
cmp X Y
```

2. Непосредственно выполнение перехода (инструкции для сравнения беззнаковых целых).

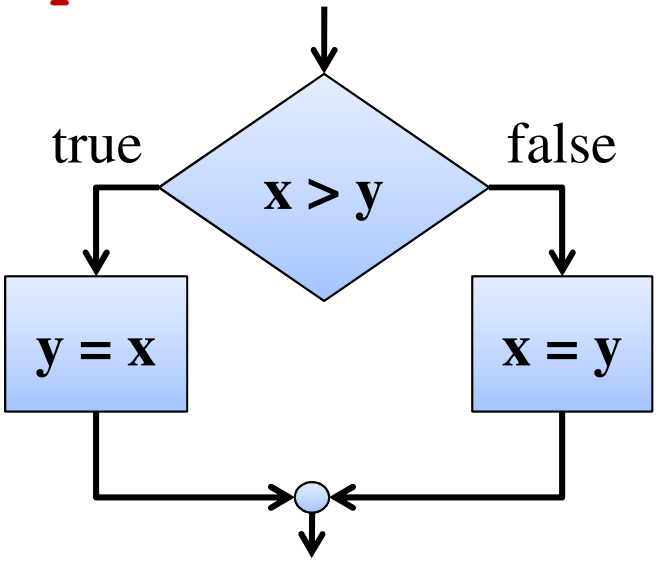
JA	(Jump if Above) переход, если X > Y
JB	(Jump if below) переход, если X < Y
JBE	(Jump if Below or Equal) переход, если X ≤ Y
JAE	(Jump if Above or Equal) переход, если X ≥ Y
JE	(Jump if equal) переход X = Y

Реализация ветвления на языке ассемблера

```
main:
   . . .
   movl   $15, -4(%ebp) // x = 15
   movl   $10, -8(%ebp) // y = 10
   movl   -4(%ebp), %eax // eax = x
   cmpl   -8(%ebp), %eax // x - y
   jle .L2// x   ≤   y
   movl   -8(%ebp), %eax
   movl   %eax, -4(%ebp)
   jmp .L5
.L2:
   movl   -4(%ebp), %eax
   movl   %eax, -8(%ebp)
.L5:
   . . .
```

false

true



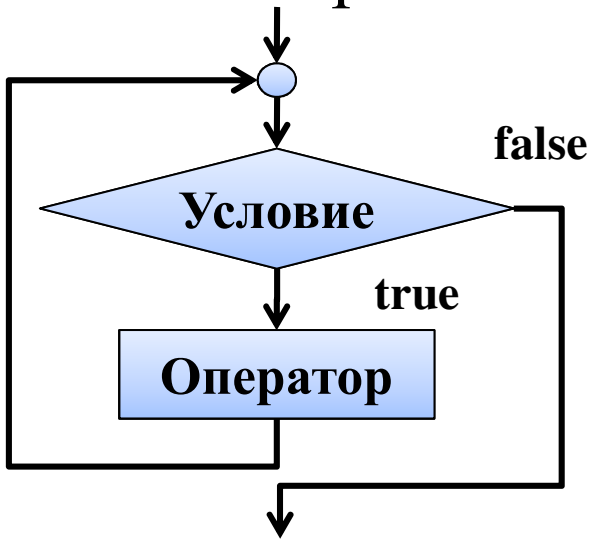
Условный переход в обратном направлении

Операция условного перехода не накладывает никаких ограничений на его направление. То есть переход может осуществляться как в прямом направлении (конструкция ветвления), так и в обратном (на инструкцию, имеющую меньший адрес).

0x1000	instr #1	
0x1004	jmp 0x1014	Пропуск тела для проверки
0x1008	instr #2	Тело цикла
0x100c	instr #3	
0x1010	instr #4	
0x1014	cmp %eax %ebx	Вычисление условия
0x1018	jle 0x1008	Продолжение/выход
0x101c	instr #5	Инструкция после цикла

Реализация цикла на языке ассемблера

```
"while" "(" Условие ")"
    Оператор.
```




```
#include <stdio.h>
int main()
{
  int i = 0;
  while( i < 20 ){
    i++;
  }
  return 0;
}
```

```
    movl   $0, -4(%ebp)
    jmp .L2
.L3:
    addl   $1, -4(%ebp)
.L2:
    cmpl   $19, -4(%ebp)
    jle .L3
```

JLE	(Jump if Less than or Equal) переход, если X ≤ Y