Введение.

Язык Си — это стандартизированный процедурный язык программирования, разработанный в 1972 году сотрудником компании AT&T Bell Laboratories Деннисом Ритчи (англ. Dennis Ritchie) в качестве развития языка Би. Он был создан как основа при проектировании операционной системы UNIX. Сейчас язык Си используется во множестве операционных систем и является самым распространенным и широко используемым языком программирования (по данным на 2013 год).

Несмотря на то, что изначально язык Си не разрабатывался как язык для обучения программированию (в отличие, например, от Pascal и Qbasic), он активно используется в этом направлении. Синтаксис этого языка стал основой для множества других языков программирования.

Особенностями этого языка являются:

- небольшое количество ключевых слов
- наличие сложных типов данных (структуры, объединения)
- возможность использования указателей, работа с памятью
- наличие внешних стандартных библиотек
- компиляция программного кода в бинарный код
- использование макропроцессора

Сравнение с другими языками:

- язык является прародителем C++, Objective C, C#
- язык сильно повлиял на Java, Perl, Python
- низкий уровень языка Си предоставляет выигрыш в скорости исполнения кода
- недостатки по сравнению с другими языками:
  - отсутствие исключений (exceptions)
  - отсутствие проверки диапазонов (range-checking)
  - отсутствие автоматической сборки мусора
  - Си не является объектно-ориентированным языком
  - отсутствие полиморфизма

История развития языка программирования Си:

1972 — разработка языка Си

1978 — опубликование языка Си; появление первой спецификации

1989 — появление стандарта C89 (известен как ANSI C или Standard C)

1990 — ANSI C, адаптированный под ISO, известен как стандарт C90

1999 — появление стандарта C99 (этот стандарт имеет обратную совместимость со стандартом C89 и полностью реализован во многих компиляторах)

2011 — выход стандарта C11 (добавлена поддержка многопоточности)

В данном пособии будет использован синтаксис ANSI / ISO C (C89/C90)

Алфавит языка Си.

В алфавит входят:

• прописные и строчные буквы латинского алфавита A-Z, a-z;

• цифры 0-9;

• специальные знаки: . , ; : ? ! " „ + - * / % ( ) [ ] { } < = > \ & # _ ~ ^;

• символы пробела: пробел, табуляция, символ конца строки.

Таблица 1. Наименования символов языка Си.

Символ	Наименование	Символ	Наименование
,	запятая	+	плюс
.	точка	-	минус (дефис)
;	точка с запятой	( )	круглые скобки
:	двоеточие	{ }	фигурные скобки
?	вопросительный знак	[ ]	квадратные скобки
!	восклицательный знак	<	меньше
'	одинарная кавычка (апостроф)	>	больше
|	вертикальная черта	#	решетка
/	дробная черта (слеш)	%	процент
\	обратная черта (обратный слеш)	&	амперсанд

Лексемы

Лексемы — это наименьшие неделимые элементы языка, из которых составляются остальные конструкции.

Классы лексем:

- ключевые слова
- идентификаторы
- константы
- строковые литералы
- операторы
- разделители и пунктуаторы

Ключевые слова зарезервированы в качестве служебных слов и не могут использоваться в другом смысле:

Ключевые слова языка Си:

```
auto     double      int         struct      break
else     long        switch      register    typedef
char     extern      return      void        case
float    unsigned    default     for         signed
union    do          if          sizeof      volatile
continue             enum        short       while
```

Идентификаторы — это символические имена, которыми обозначают переменные, функции, типы данных, метки и другие объекты. Идентификатор может состоять из латинских букв (прописные и строчные), цифр и символа подчеркивания. При этом цифра не может быть первым символом идентификатора.

Язык Си, в отличие от некоторых других языков, чувствителен к регистру. Это значит, что, например, идентификаторы `variable`, `Variable` и `VariAble` являются тремя различными идентификаторами.

Константы служат для представления постоянных, неизменяемых значений.

Строковые литералы — это любая последовательность символов, которая включена в двойные кавычки.

Операторы — это знаки арифметических операций или операций сравнения: `+ ++ = == > < >= <=` .

Разделители и пунктуаторы осуществляют функции группировки и упорядочения кода (`* = ( ) [ ] { } , ; : … #`).

Комментарии

Многострочные комментарии

Комментарий для нескольких строк начинается с символов `/*` и заканчивается символами `*/`. Между звездочкой и слешем не должно быть никаких пробелов. Любой текст, расположенный между начальными и конечными символами комментария, компилятором игнорируется. Например,

```
/* это
многострочный
комментарий */
```

Комментарии могут находиться в любом месте программы, за исключением середины ключевого слова или идентификатора. Кроме того, комментарии не следует размещать и в середине выражений, потому что так труднее разобраться и с выражениями, и с самими комментариями.

Многострочные комментарии не могут быть вложенными. То есть в одном комментарии не может находиться другой.

Однострочные комментарии

Такой комментарий начинается с символов `//` и заканчивается в конце строки. Например:

```
// это однострочный комментарий
```

Однострочные комментарии особенно полезны тогда, когда нужны краткие, не более чем в одну строку пояснения.

Однострочный комментарий может находиться внутри многострочного комментария. Например, следующий комментарий является вполне допустимым:

```
/* это // вложенный однострочный комментарий
в многострочный коментарий */
```

Комментарии должны находиться там, где требуется объяснить работу кода. Например, в начале всех функций могут быть комментарии, которые сообщают, что именно делает функция, как она вызывается и что возвращает. Чем сложнее ваша программа, тем больше комментариев она требует.

Кроме обычного назначения, комментарии часто помогают при выявлении ошибок в коде. Закомментировав ту или иную часть кода, можно локализовать место ошибки и устранить ее.

Типы данных

Простые базовые типы данных включают в себя несколько разновидностей

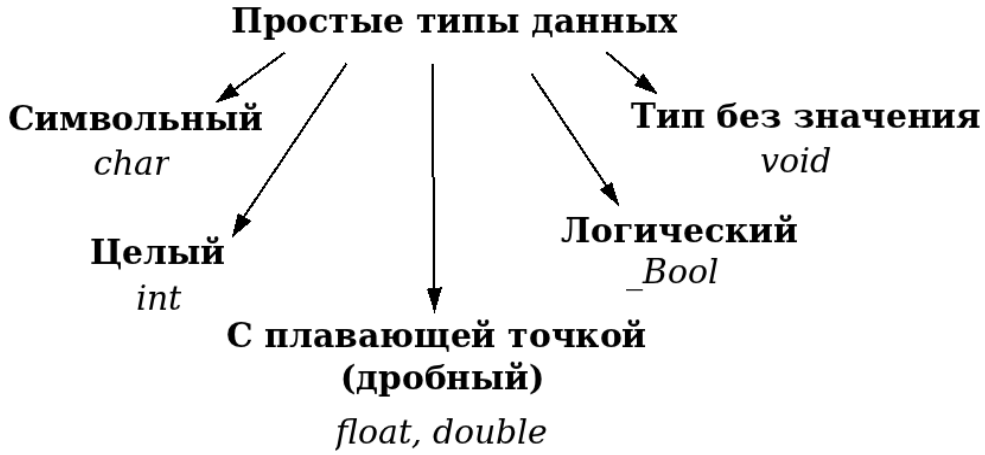


Рисунок 1. Простые типы данных языка Си.

Символьный тип (`char`) содержит данные в виде единичных символов, букв, цифр или знаков. При написании кода программы единичные символы заключают в одинарные кавычки.

Числовые типы (`int, float, double`) включают в себя целые и дробные числа.

`int` — целые числа
`float` — дробные числа (одинарная точность)
`double` — дробные числа (двойная точность)

Логический тип (`_Bool`) позволяет хранить значения:

- `1` (`true`)
- `0` (`false`)

Тип без значения (`void`) служит для объявления функции, которая не возвращает значения, или для создания универсального указателя.

Модификация простых базовых типов

Базовые типы данных (кроме `void` и `_Bool`) могут иметь спецификаторы длины и спецификаторы знака, предшествующие им в тексте программы (`signed, unsigned, short, long`). Спецификатор так изменяет значение базового типа, что он более точно соответствует своему назначению в программе.

Символьный и целый типы могут быть знаковыми (`signed`) или беззнаковыми (`unsigned`). Целые числа со знаком и без знака отличаются интерпретацией нулевого бита числа. Если целая переменная объявлена со знаком, то компилятор считает, что нулевой бит содержит знак числа. Если в нулевом бите записан 0, число считается положительным, а если 1 — отрицательным.

Спецификаторы длины (`short, long`) определяют диапазоны значений типа и сколько оперативной памяти будет выделено для хранения переменных данного типа.

На основе базовых типов можно образовать сколь угодно сложные и многоуровневые типы данных:

- Массивы
- Структуры
- Перечисления
- Объединения
- Битовые поля

Эти типы будут рассмотрены позднее.

Хранение данных. Переменные и константы

Переменные

Для хранения данных используются переменные. Переменная — это идентификатор, скрывающий за собой область памяти с хранящимися там данными. Иначе говоря, это имя области памяти.

У каждой переменной есть тип, который соответствует тому, какой тип данных хранит переменная. Данные, которые хранит переменная (значение переменной), могут изменяться в пределах типа переменной. Это значит, что если переменная хранит данные одного типа, то она никогда не сможет начать хранить данные другого типа.

Имя переменной может состоять из записанных в любом порядке символов латинского алфавита, цифр и символа подчеркивания. При этом первым символом имени переменной не может быть цифра.

При выборе имен переменных желательно придерживаться правила о том, что переменные должны называться в соответствии со своим смыслом. Например, если переменная хранит «некую сумму», то логичнее было бы назвать ее `sum`, `amount`, `count` или `summa`, нежели просто `x`, `k` или `s1`. Для имен переменных рекомендуется использовать английские слова или в крайнем случае транслитерацию русских слов.

При написании программы очень важно стараться употреблять такие конструкции, которые будут понятны не только вам, но и другим программистам. Представьте, что было бы, если бы авторы писали свои книги неразборчивым почерком, может быть, даже наискосок или вверх ногами. Такие книги мало кому бы захотелось и удалось бы прочитать, и тем более понять. Тоже самое можно сказать о программах: ответственно относитесь к своим программам, пишите их аккуратно, соблюдая сложившиеся правила. Большинство программистов работают в команде, и написание понятного кода — это важнейшее условие плодотворного взаимодействия.

Объявление переменных

Перед тем как использовать переменные в программе, их нужно объявить. Во время объявления переменных происходит выделение памяти для их хранения. Количество памяти соответствует типу переменных.

Объявление переменных может быть расположено в трех местах программы: внутри функции, вне всех функций и при определении параметров функции. Это места объявлений соответственно локальных переменных, глобальных переменных и формальных параметров функций.

При объявлении переменных сначала записывается тип данных, который будет содержать переменная, затем через пробел - имя переменной. Если

требуется объявить несколько переменных одного типа, то это можно сделать через запятую.

Для инициализации переменной после ее объявления нужно поставить знак равенства и указать начальное значение. Неинициализированные локальные переменные до первого присвоения имеют произвольное значение. Неинициализированные глобальные переменные в начале работы программы автоматически обнуляются.

Глобальные переменные инициализируются только один раз в начале работы программы. Локальные переменные инициализируются каждый раз при входе в блок (область между ближайшими фигурными скобками), в котором они объявлены.

Следует сказать о том, что приобретение привычки всегда инициализировать переменные поможет вам избежать ошибок в программе. Недопустимо, чтобы переменные содержали мусор на любом этапе их жизни.

Объявление переменных

```
тип_данных  имя_переменной [= начальное_значение];
```

Примеры объявления переменных

По одной переменной:

```
int x;
double y;
char letter;
```

По несколько переменных сразу:

```
int i, k, age, mass;
double cat, ball, f, weight;
char a, b, word, some_thing;
```

Объявление с инициализацией всех или только некоторых переменных:

```
int y = 5, apple = 9;
double var, age_kitten = 1.7, g = 2.467;
char k, letter = 'h', word = '?', name;
```

Константы

Константа — это фиксированное значение, которое не может быть изменено программой. Она может относиться к любому базовому типу.

Константы выполняют несколько важных функций:

- использование констант как размерности массивов (размерность массива не может быть переменной). (Этот способ употребления констант может не работать при использовании некоторых версий компилятора.)
- слежение за тем, чтобы случайным образом не изменить величину какой-либо переменной
- для удобства пользователя и разграничения числовых значений в программе

Квалификатор `const`

Для того, чтобы обозначить, что переменная является константой, нужно добавить квалификатор **`const`** перед объявлением типа переменной. Присваивать значение константе можно только при объявлении.

```
const тип_константы имя_константы  = значение;
```

Например, здесь

```
const int  mount = 10;
```

создается переменная с именем `mount` и ей присваивается начальное значение `10`, которое в дальнейшем в программе изменить никак нельзя. Переменную, к которой в объявлении применен квалификатор **`const`**, можно использовать в различных выражениях справа от знака присвоения как и обычную переменную.

Примеры объявления констант

```
const int a = 11, b = 16;
const double dlina = 3.9, shirina = 67.45;
const char word = '}';
```

```
const char letter = word;
```

Директива #define

Директива **#define** определяет идентификатор и последовательность символов, которая во время компиляции программы будет подставляться вместо идентификатора каждый раз, когда он встретится. Идентификатор называется макросом, а сам процесс замены — макрозаменой. В общем виде директива выглядит таким образом:

```
#define имя_макроса последовательность_символов
```

Обратите внимание, что в этом выражении в конце нет точки с запятой. Если попробовать поставить точку с запятой в конце строки с директивой **#define**, то это может привести к ошибке, так как при макрозамене вместо макроса подставится вся последовательность символов до конца строки (вместе с точкой с запятой).

Имена макросов часто используются для определения имен так называемых "магических чисел". Рекомендуется избегать употребление вообще каких-либо чисел в программах и заменять их макросами (**#define**) или константами (**const**). Например, если в нашей программе предполагается использовать отрезок `[10, 20]`, то вместо значения `10` можно использовать слово `LEFT`, а вместо значения `20` — слово `RIGHT`. Тогда можно сделать следующие объявления с помощью директивы **#define**:

```
#define LEFT 10
#define RIGHT 20
```

В результате компилятор будет подставлять `10` или `20` каждый раз, когда в вашем файле исходного кода встречается идентификатор соответственно `LEFT` или `RIGHT`.

Например, следующий код выводит на экран **20 10 200**:

```
printf("%d %d %d", RIGHT, LEFT, LEFT + 190);
```

После определения имени макроса его можно использовать в определениях других имен макросов. Вот, например, код, определяющий три макроса:

```
#define ONE 1
#define TWO ONE + ONE
#define THREE ONE + TWO
```

Макрозамена — это замена какого-либо идентификатора связанной с ним последовательностью символов. Поэтому, если требуется определить стандартное сообщение об ошибке, то можно написать следующее:

```
#define STD_ERROR "стандартная ошибка при вводе\n"
/*  ----несколько
    строчек
    кода--------   */
printf(STD_ERROR);
```

Теперь каждый раз, когда встретится идентификатор `STD_ERROR`, компилятор будет его заменять строкой `"стандартная ошибка при вводе\n"`. После макрозамены для компилятора выражение `printf(STD_ERROR);` на самом деле будет выглядеть таким образом:

```
printf("стандартная ошибка при вводе\n");
```

Если идентификатор находится внутри строки, заключенной в кавычки, то замены происходить не будет. Например, при выполнении кода

```
#define VALIDATION "Это Проверка"
/*  ----несколько
    строчек
    кода--------   */
printf("VALIDATION");     //первый случай (в кавычках)
/*  ----еще несколько
    строчек
    кода--------   */
printf(VALIDATION);       //второй случай (без кавычек)
```

в первом случае вместо сообщения: "Это Проверка" будет выводиться сообщение: "VALIDATION", а во втором случае произойдет макрозамена, и на экране появится сообщение "Это проверка".

Если последовательность символов не помещается в одной строке, то ее можно продолжить на следующей строке, поместив в конце предыдущей обратную косую черту (обратный слеш):

```
#define    LONG_STRING     "это пример очень \
                           длинной строки"
```

Программисты, пишущие программы на языке Си, в именах, определяемых идентификатором **#define**, используют буквы верхнего регистра. Если следовать этому правилу, то тот, кто будет читать программу, с первого взгляда сможет понять, что будет происходить макрозамена. Кроме того, все директивы **#define** лучше всего помещать в самом начале файла, а не разбрасывать по всей программе.

Операции и Операторы

Язык Си содержит большое количество встроенных операций. Их роль значительно больше, чем в других языках программирования. Существует четыре основных класса операций: арифметические, логические, поразрядные и операции сравнения. Кроме них есть также некоторые специальные операторы, например оператор присваивания.

Оператор

Любое выражение, которое заканчивается точкой с запятой, является оператором.

Примеры операторов:

```
func(); /* вызов функции */
a = b + c; /* оператор присваивания */
b++; /* оператор инкремента */
; /* пустой оператор */
```

Первый оператор выполняет вызов функции, второй — присваивание. Третий оператор увеличивает значение переменной на единицу. Последний оператор — это пустой оператор, который не выполняет никаких действий.

Блок операторов

Блок операторов — это последовательность операторов, заключенных в фигурные скобки, которые рассматриваются как одна программная единица. Операторы, составляющие блок, логически связаны друг с другом. Блок всегда начинается открывающейся фигурной скобкой { и заканчивается закрывающейся }. Чаще всего блок используется как составная часть какого-либо оператора, выполняющего действие над группой операторов, например, **if** или **for**. Однако блок можно поставить в любом месте, где может находиться оператор.

```
int x = 0, y = 5, z = -3;
{// это блок операторов
    x = y + z;
    printf("результат сложения равен %d", x);
}
```

```
for (x = 0; x < 5; x += 2) {
// это тоже блок операторов
     printf("четное число %d\t", x);
     printf("нечетное число %d\n", x + 1);
}
```

Оператор присваивания

Оператор присваивания может присутствовать в любом выражении языка Си. Этим Си отличается от большинства других языков программирования, в которых присваивание возможно только в отдельном операторе.

Общая форма оператора присваивания:

```
имя_переменной = выражение;
```

Выражение может быть просто константой или сколь угодно сложным выражением. Оператором присваивания служит знак "=". Адресатом (получателем, левой частью оператора) присваивания должен быть объект, способный получить значение, например переменная.

Множественные присваивания

В одном операторе присваивания можно присвоить одинаковое значение многим переменным. Для этого используется оператор множественного присваивания:

```
x = y = z = 0;
```

При этом операции присваивания выполняются справа налево. Таким образом, сначала выполнится операция `z = 0`, затем `y = z`, и, наконец, `x = y`.

Арифметические действия над переменными.

Таблица 2. Арифметические операции языка Си.

Оператор	Операция	Класс операций	
++	инкремент (a++)	аддитивные	унарные
--	декремент (a--)		
+	сложение (a + b)		бинарные
-	вычитание (a - b)		
-	унарный минус (- a)	мультипликативные	унарная
*	умножение (a * b)		бинарные
/	деление (a / b)		
%	остаток от деления (a % b)		

Арифметические операции можно разделить на унарные, бинарные, аддитивные и мультипликативные.

Унарные арифметические операции производятся с использованием только одной переменной.

Операции называются бинарными, если в них участвуют две переменные

Аддитивные арифметические операции — это операции сложения и вычитания.

Мультипликативные операции — это операции умножения, деления и нахождения остатка от деления целого числа на целое число.

Нахождение остатка от деления `%` в языке Си работает так же, как и в других языках, его результатом является остаток от целочисленного деления.

```
int x = 5, y = 2;
int z = x % y;
```

Здесь переменная `z` будет равна 1, так как при делении 5 на 2 получается 2 целых и 1 в остатке. Этот оператор, однако, нельзя применять к типам данных с плавающей точкой.

Составное присваивание

Составное присваивание — это разновидность оператора присваивания, в которой запись сокращается и становится более удобной в написании. Например, оператор

`a = a + 35;` можно записать как `a += 35;`

Оператор `"+="` сообщает компилятору, что к переменной `a` нужно прибавить 35.

Составные операторы присваивания существуют для всех бинарных операций (то есть операций, имеющих два операнда). Любой оператор вида

```
переменная = переменная оператор выражение;
```

можно записать как

```
переменная оператор= выражение;
```

`x = x + 3;`		`x += 3;`
`x = x - 3;`		`x -= 3;`
`x = x * 3;`	то же самое, что и	`x *= 3;`
`x = x / 3;`		`x /= 3;`
`x = x % 3;`		`x %= 3;`

Составное присваивание значительно компактнее, чем соответствующее простое присваивание, поэтому очень часто встречается в текстах программ.

Инкремент и Декремент

Кроме составного присваивания следует рассмотреть два очень полезных оператора для упрощения часто употребляемых операций. Это инкремент `++` (плюс-плюс) и декремент `--` (минус-минус). Оператор инкремента увеличивает значение операнда на `1`, а декремента - уменьшает на `1`.

`x = x + 1;` можно записать как `++x;` или `x++;`

аналогично

`x = x - 1;` равносилен оператору `--x;` или `x--;`

Инкремент и декремент могут иметь префиксную (`++x; --x;`) и постфиксную (`x++; x--;`) форму, то есть могут предшествовать операнду (целой переменной) или следовать за ним. Если оператор инкремента или декремента предшествует операнду, то сама операция выполняется до использования результата в выражении. Если же оператор следует за операндом, то в выражении значение операнда используется до выполнения операции инкремента или декремента. То есть для выражения эта операция как бы не существует, она выполняется только для операнда. Например,

```
x = 10;
y = ++x;
```

здесь сначала переменной `x` присваивается значение `11`, а затем переменной `y` присваивается значение 11. Однако если написать

```
x = 10;
y = x++;
```

,то сначала переменной `y` будет присвоено значение `10`, а потом уже прибавится единица к переменной `x`. В обоих случаях переменной `x` будет присвоено значение `11`, разница только в том, когда именно это случится: до или после присваивания значения переменной `y`.

Большинство компиляторов языка Си генерируют для инкремента и декремента очень быстрый, эффективный, объектный код, значительно лучший, чем для соответствующих операторов присваивания. Поэтому везде, где это возможно, рекомендуется использовать инкремент и декремент.

Стандартная библиотека математических функций `math.h`.

В стандартную библиотеку включено множество функций для математических вычислений. Вот некоторые из них:

Таблица 3. Некоторые математические функции библиотеки math.h

Математический вид	**Библиотека math.h**	**Описание функции**
$\|x\|$	abs(x)	модуль целого числа
$\|x\|$	fabs(x)	модуль дробного числа
$\sqrt{x}$	sqrt(x)	квадратный корень
a^x	pow(a, x)	возведение в степень
e^x	exp(x)	экспонента
ln(x)	log(x)	натуральный логарифм
lg(x)	log10(x)	десятичный логарифм
sin(x)	sin(x)	синус
cos(x)	cos(x)	косинус
tan(x)	tan(x)	тангенс
--	round(x)	округление до целого
--	ceil(x)	округление до большего целого
--	floor(x)	округление до меньшего целого

Стоит заметить, что тригонометрические функции принимают значения не в градусах, а в радианах. Например, `cos(60)` не будет равен `0.5`, потому что мы посчитали синус угла `60` радиан, а не `60` градусов.

Стандартная библиотека `stdlib.h`. Генерация псевдослучайных чисел.

Псевдослучайные числа (далее - случайные числа) генерируются с помощью функции **`rand()`**. Эта функция возвращает целое число в диапазоне от нуля до `RAND_MAX`. Случайное число генерируется алгоритмом, который возвращает последовательность внешне не связанных цифр при каждом вызове. Этот алгоритм использует серии вызовов, которые должны быть инициализированы значением с помощью функции **`srand()`**.

`RAND_MAX` - это константа, определенная в библиотеке `<stdlib.h>`. Ее значение может варьироваться в зависимости от используемой реализации языка Си, но она не меньше, чем `32767`.

Рассмотрим несколько примеров для понимания принципов работы со случайными числами и функцией **`rand()`**.

Пример 1:

Сгенерировать целое случайное число от `0` до `9`.

```
int x = rand() % 10;
```

Функция **`rand()`** генерирует число от нуля до `RAND_MAX`, а затем находится остаток от деления этого числа на `10`. Рассмотрим, почему при этом образуется диапазон целых чисел от `0` до `9`.

При нахождении остатка от деления одного числа на другое результатом могут быть только определенные целые числа, диапазон этих чисел строго ограничен. Например, если какое-либо число целочисленно делится на `3`, то остатком от деления могут быть только три числа:

- ноль (если делится без остатка)
- один (если в остатке единица)
- два (если в остатке двойка)

Других чисел в остатке быть не может.

Соответственно, если находить остаток от деления на `10`, то в результате можно получить все целые числа в диапазоне от `0` до `9`, вне зависимости от того, какое число мы делим на `10`. Таким образом, какое бы число не сгенерировала функция **`rand()`** при нахождении остатка от деления на `10`, получится одно из чисел диапазона `[0; 9]`

Пример 2:

Сгенерировать целое число в диапазоне от -328 до 1485.

```
int x = -328 + rand() % 1814;
```

Сначала подсчитаем, сколько чисел нам нужно сгенерировать. 328 чисел в отрицательную сторону по оси абсцисс, плюс еще 1485 чисел в положительную сторону и плюс число ноль. Всего получается, что заданный диапазон содержит 1814 чисел. Такое количество чисел можно сгенерировать, если найти остаток от деления результата функции rand() на 1814 (rand() % 1814). Так получатся числа от нуля до 1814. Чтобы получить нужный диапазон [-328; 1485], нужно сдвинуть полученный диапазон [0; 1814] на -328. Получаем -328 + rand() % 1814.

Вторым способом подсчитать количество цифр между двумя любыми числами является вычитание из большего числа меньшего числа и прибавление единицы.:

```
int x = -328 + rand() % (1485 - (-328) +1);
```

Пример 3:

Сгенерировать дробное случайное число в диапазоне от нуля до единицы.

```
double x = rand() / (double)RAND_MAX;
```

Для генерации дробных чисел используется обычное деление. В данном случае деление на число RAND_MAX, приведенное к дробному типу, для избежания целочисленного деления. Функция rand() генерирует случайное число в диапазоне от 0 до RAND_MAX. Если мы разделим его на RAND_MAX, то минимально получаем ноль (0 / RAND_MAX) а максимально - единицу (RAND_MAX / RAND_MAX).

Инициализация генератора случайных чисел.

Функция srand() отвечает за инициализацию генератора случайных чисел. Генератор случайных чисел инициализируется с помощью аргумента, передаваемого в качестве параметра функции. Если параметром задавать одно и тоже число, то при различных запусках программы происходит генерация случайных чисел в одинаковом порядке. Например, если в 1й запуск была сгенерирована последовательность 38, 2, -9.., то и во все последующие запуски будет сгенерирована та же последовательность.

Для того, чтобы при каждом запуске последовательности случайных чисел были разными, необходимо, чтобы при каждом запуске программы параметрами были различные числа. Для этого можно использовать функцию time() из стандартной библиотеки <time.h>. Эта функция возвращает

текущее календарное время системы (для UNIX систем это количество секунд, прошедших с полночи 1 января 1970 года по Гринвичу), которое при каждом запуске будет различным. Функция **time**() так же должна содержать в себе параметр, в качестве которого обычно используется ноль.

Таким образом, получаем, что вызов функции **srand**() выглядит так:

```
srand(time(0));
```

Функция **srand**() может вызываться в программе несколько раз. При этом генератор случайных чисел будет инициализироваться заново. Например, следующий код выведет на экран три целых числа, первое и последнее из которых будут одинаковыми, так как эти числа генерируются с одинаковым параметром инициализации генерации случайных чисел, и они одинаковые по порядку генерации.

```
srand(1);
printf("First number: %d\n", rand() % 100);
srand(time(0));
printf("Random number: %d\n", rand() % 100);
srand(1);
printf("Again the first number: %d\n", rand() % 100);
```

Явное и неявное преобразования типов

В Си различают явное и неявное преобразование типов данных. Неявное преобразование типов данных выполняет компилятор, а явное преобразование данных выполняет сам программист.

Неявное преобразование типов при вычислениях

При неявном преобразовании типов данных результат любого вычисления будет преобразовываться к наиболее точному типу данных из тех, которые участвуют в вычислении.

Например, если вы делите целое число на другое целое число, то в ответе вы тоже получаете целое число. Но если хотя бы одно из чисел (делитель или делимое) является дробным числом, то в результате получается дробное число, так как оно обладает большей точностью.

Например,

```
int x = 5, y = 2;
int z = x / y;          // (z = 2)
```

здесь `z = 2`, так как это целочисленное деление, и вся дробная часть отбрасывается.

```
int x = 5, y = 2;
double z = x / y;          // (z = 2.0)
```

здесь `z = 2.0`, так как сначала происходит целочисленное деление, а затем преобразование целого получившегося числа `2` в дробное число `2.0` для записи в дробную переменную `z`.

Целое число делим на дробное:

```
int    x = 5;
double   y = 2.0;
double   z = x / y;    // (z = 2.5)
```

Дробное число делим на целое:

```
int    y = 2;
double   x = 5.0;
double   z = x / y; //(z = 2.5)
```

Дробное число делим на дробное:

```
double   x = 5.0;
double   y = 2.0;
double   z = x / y;//(z = 2.5)
```

Во всех трех случаях значение дробной переменной `z` получается одинаковым `z = 2.5`, так как в процессе деления участвует хотя бы одно дробное число.

Неявные преобразования типов при операции присваивания

Неявное преобразование типов при присваивании происходит тогда, когда справа и слева от операнда присваивания стоят переменные или выражения различных типов. В этом случае выражение с правой стороны операнда неявно преобразуется к типу выражения с левой стороны.

Например,

```
double f = 50;
```

здесь дробной переменной `f` присваивается значение целого числа `50`. При этом число `50` преобразуется компилятором языка в дробное число `50.0`, а затем происходит операция присваивания.

```
int var1 = -34;
double var2 = var1;
```

Здесь целое значение переменной `var1` преобразуется компилятором в дробное значение `-34.0` (при этом переменная `var1` остается целого типа, и в ней ничего не меняется). А затем происходит присвоение переменной `var2` дробного значения `-34.0`.

```
double age = 9.8;
int class = age - 7;
```

Если целой переменной присвоить дробное значение, то произойдет отбрасывание дробной части. Здесь сначала выполнится вычитание `age – 7`. Результат такого выражения будет дробным `2.8`. Перед присвоением отбрасывается дробная часть, и получается целое число `2`, которое присваивается переменной `class`.

```
double u = 7.34;
const int p = u = 90;
```

здесь дробной переменной `u` присвоили дробное значение `7.34`. Затем этой же переменной присвоили целое значение `90`, предварительно преобразовав его в дробное `90.0`. Теперь `u = 90.0`. И, наконец, целой переменной `p` присвоили дробное значение `90.0`, предварительно преобразовав его в целое значение `90`. Теперь `p = 90`.

Явное преобразования типов.

Для того, чтобы произвести явные преобразования типа переменной или результата вычислений, используются операции преобразования типа:

```
(тип, к которому нужно преобразовать)  выражение
```

```
double x;
x = (double)5;
```

Целое число 5 сначала приводится к дробному 5.0, а затем это значение присваивается переменной `x` (`x = 5.0`). Тоже самое происходит, если

```
double x;
x = (double)(a + c);
```

результат сложения двух переменных преобразуется к дробному виду, а затем присваивается переменной `x`.

```
double x;
x = (int)3.74;
```

Дробное число `3.74` преобразуется к целому путем отбрасывания дробной части. Затем получившаяся величина `3` преобразуется в дробную величину `3.0`, чтобы записать ее в дробную переменную `x`. В итоге получается, что `x = 3.0`.

```
double x;
x = (int)7.9 / 2;
```

Число `7.9` преобразуется к целому типу `7` и делится на целое число `2`. В результате целочисленного деления получается целое число `3`, которое затем преобразуется в дробное `3.0` и записывается в дробную переменную `x`.

Явное преобразование типов бывает очень необходимым при различных вычислениях. Например, если вам нужно найти среднее арифметическое двух целых переменных.

```
double average = (double)(x + y) / 2;
```

Сумма двух целых чисел преобразуется в дробное число, а затем делится на 2, и дробный результат присваивается дробной переменной average.

Стоит отметить, что результатом вычислений всех математических функций (кроме `abs()`) являются дробные числа. Это стоит иметь ввиду при учете неявных преобразований типов.

Приоритет выполнения арифметических операций

В языке программирования Си есть определенный порядок выполнения арифметических операций. Он схож с тем, который используется в математике.

Приоритет выполнения операций:

1. Операции в скобках
2. Функции стандартной математической библиотеки `math.h`
3. Унарные операции
4. Умножение, деление, нахождение остатка от деления
5. Сложение, вычитание

Операции с одинаковым приоритетом выполняются слева направо. Используя круглые скобки, можно изменить порядок вычислений. В языке Си круглые скобки придают операции (или последовательности операций) наивысший приоритет. Лучше всегда явно указывать приоритет операций с помощью скобок, чтобы не получить ошибку, которую потом очень сложно будет выявить.

```
x = (a + 8 * c) * sqrt(a - c) + (c - 3 / a * d);
```

здесь три выражения в скобках

1. **a + 8 * c**
   - умножение 8 * c
   - сложение a + (результат умножения)

2. **a - c**

- вычитание **a - c**

3. **c - 3 / a * d**

- деление **3 / a**
- умножение (результат деления) *** d**
- вычитание **c -** (результат умножения)

Следующая по старшинству идет математическая функция `sqrt()`

4. **sqtr( )**

- вычисление корня **sqrt (**результат вычитания**)**

Остается три операции умножение, сложение и присваивание

5. **( ) * sqrt ( )**

- умножение (результат пункта 1) * (результат пункта 4)

6. **( ) + ( )**

- сложение (результат пункта 5) + (результат пункта 3)

7. **x = ( )**

- присваивание x = (результат пункта 6)

Операции сравнения и логические операции.

Операции сравнения — это операции, в которых значения двух переменных или выражений сравниваются друг с другом. Логические же операции реализуют средствами языка Си операции формальной логики. Между логическими операциями и операциями сравнения существует тесная связь: результаты операций сравнения часто являются операндами логических операций.

В операциях сравнения и логических операциях в качестве операндов и результатов операций используются значения `ИСТИНА` (`true`) и `ЛОЖЬ` (`false`). В языке Си значение `ИСТИНА` представляется любым числом, отличным от нуля. Значение `ЛОЖЬ` представляется нулем.

Таблица 4. Операторы сравнения и логические операторы.

Операторы сравнения	
Оператор	**Операция**
>	больше
>=	больше или равно
<	меньше
<=	меньше или равно
==	равно
!=	не равно
Логические операторы	
Оператор	**Операция**
&&	И
||	ИЛИ
!	НЕ

Приоритет логических операций

1. !
2. > >= < <=
3. == !=
4. &&
5. ||

Как операции сравнения, так и логические операции имеют низший приоритет по сравнению с арифметическими. То есть, выражение `10 > 1 + 12` интерпретируется как `10 > (1 + 12)`. Результат, конечно, равен `ЛОЖЬ`.

В одном выражении можно использовать несколько операций:

```
10 > 5 && !(10 < 9) || 3 < 4
```

в этом случае результатом будет `ИСТИНА`.

В языке Си не определена операция "исключающего ИЛИ" (exclusive OR, или XOR). Однако с помощью логических операторов можно описать функцию, выполняющую эту операцию `(a || b) && !(a && b)`.

Как и в арифметических выражениях, для изменения порядка выполнения операций сравнения и логических операций можно использовать круглые скобки. Например, выражение:

```
!0 && 0 || 0
```

равно ЛОЖЬ. Однако если добавить скобки как показано ниже, то результатом будет ИСТИНА:

```
!(0 && 0) || 0
```

Ввод/вывод данных

Стандартная библиотека **stdio.h** (**st**andar**d** **i**nput **o**utput) содержит в себе различные функции для ввода и вывода данных. Рассмотрим функции вывода на экран и ввода с клавиатуры **printf**() и **scanf**().

Функция консольного вывода printf()

Функция **printf**() выводит информацию на экран с учетом выбранного форматирования. На вход функции подается строка форматирования и список параметров.

```
printf(" строка форматирования ", список параметров);
```

Строка форматирования — это специальная последовательность символов, которая отображает, как именно вы хотите записать число или символ. Рассмотрим ее более подробно.

```
"%[флаг][ширина][.точность]тип"
```

Знак процента (%) указывает на то, что в это место строки форматирования будет вставлен один из параметров.

Флаги

Таблица 5. Флаги строки форматирования.

Знак	Название	Значение	Отсутствие знака
-	дефис	Выровнять по левому краю	По правому краю
+	плюс	Указать знак числа	Указать знак только для отрицательных чисел
␣	пробел	Поместить перед результатом пробел	--
0	ноль	Дополнить нулями до ширины поля	Дополнить пробелами

Ширина

Ширина — это ширина выделяемого поля для записи. Если выделяется недостаточное по ширине поле, то оно автоматически расширяется до нужного размера. Например, если для записи числа 4711 выделяется поле шириной в 1, 2 или 3, то ширина автоматически увеличивается до 4.

Точность

Точность соответствует количеству знаков после запятой, которые нужно вывести на экран. При этом происходит округление до нужного знака. Например, если вы выводите число 2.385 с точностью до двух знаков, то на экране появится число 2.39.

Тип

Тип — это условное обозначение, которое соответствует тому типу переменных, которые мы выводим на экран.

Таблица 6. Обозначения типа аргумента в строке форматирования.

Обозначение	**Значение**	**Тип аргумента (переменной)**
`d`	**`d`**`ecimal`	указатель на **`int`**
`f`	**`f`**`loat`	указатель на **`float`**
`lf`	**`l`**`ong` **`f`**`loat`	указатель на **`double`**
`c`	**`c`**`har`	указатель на **`char`**
`s`	**`s`**`tring`	указатель на `строку`

Список параметров

В список параметров могут входить символы, строки, целые и дробные числа, переменные, выражения.

Ширина и точность — это необязательные параметры. Если они не указаны, то срабатывает форматирование по умолчанию. Например для целых чисел это означает, что будет выделена минимально допустимая ширина поля, а для дробных чисел — что точность устанавливается на `6` знаков после запятой, а ширина поля так же становится минимально возможной.

Примеры вывода на экран целых чисел:

1	**printf** ("%d", 3);		3							
2	**printf** ("%4d", 3);					3				
3	**printf** ("%-4d", 3);		3							
4	**printf** ("%+7d", 3);							+	3	
5	**printf** ("%+-3d", 3);		+	3						
6	**printf** ("%0+5d", 3);		+	0	0	0	3			
7	**printf** ("%0+-5d", 3);		+	3						
8	**printf** ("%2d", 3675);		3	6	7	5				

1. В первом случае мы указываем только тип данных, которые мы выводим, поэтому все остальные параметры выбираются по умолчанию: выравнивание по правому краю, знак указывать только для отрицательных чисел, ширина поля равна количеству цифр.
2. Указана только ширина поля, значит все остальные параметры выбираются по умолчанию. Получаем параметры форматирования: выравнивание по правому краю, знак указывать только для отрицательных чисел, ширина поля равна 4.
3. Флаг «дефиса» указывает на выравнивание по левому краю. Форматирование: выравнивание по левому краю, знак указывать только для отрицательных чисел, ширина поля равна 4.
4. Флаг «плюс» требует выводить знак числа для положительных и отрицательных чисел. Форматирование: выравнивание по правому краю, знак указывать для всех чисел, ширина поля равна 7.
5. Сочетание флага «дефис» и «плюс». Форматирование: выравнивание по левому краю, знак указывать для всех чисел, ширина поля равна 3.
6. Флаг «ноль» заполняет пустые пробелы в начале нулями. Форматирование: выравнивание по правому краю, знак указывать для всех чисел, лишние пробелы заполнить нулями, ширина поля равна 5.
7. Флаги «плюс», «дефис», «ноль». При сочетании флагов «дефис» и «ноль» флаг «ноль» не учитывается, так как при выравнивании по левому краю нет лишних пробелов в начале. Форматирование: выравнивание по левому краю, знак указывать для всех чисел, ширина поля равна 5.
8. Если указанная ширина поля меньше, чем минимально необходимое поле для записи числа, то оно расширяется. В нашем случае поле указано равным 2, а число состоит из 4х цифр, поэтому ширина поля автоматически становится равной 4. Форматирование: выравнивание по правому краю, знак указывать только для отрицательных чисел, ширина поля равна 4.

Примеры вывода на экран дробных чисел (`double`):

1	**printf** ("%lf", 2.157);		2	.	1	5	7	0	0	0	
2	**printf** ("%5.1lf", 2.157);				2	.	2				
3	**printf** ("%5lf", 2.157);		2	.	1	5	7	0	0	0	
4	**printf** ("%.1lf", 2.157);		2	.	2						
5	**printf** ("%7.5lf",2.157);		2	.	1	5	7	0	0		
6	**printf** ("%+6.2lf", 2.157);			+	2	.	1	6			
7	**printf** ("%-7.0lf", 2.157);		2								
8	**printf** ("%+07.2lf", 2.157);		+	0	0	2	.	1	6		
9	**printf** ("%+-07.2lf", 2.157);		+	2	.	1	6				

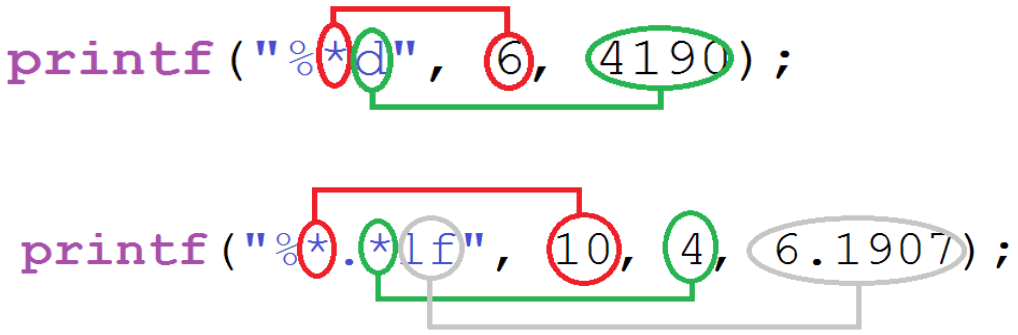
1. Если в строке форматирования дробного числа указан только тип, то по умолчанию устанавливается: выравнивание по правому краю, знак указывать только для отрицательных чисел, точность — 6 знаков после запятой, ширина поля равна количеству цифр в целой части числа плюс еще 7 (6 — для дробной части и 1 для запятой).
2. Если точность меньше, чем количество знаков после запятой, то число округляется до указанной точности. Форматирование: выравнивание по правому краю, знак указывать только для отрицательных чисел, точность равна 1, ширина поля равна 5.
3. Если указана только ширина поля, то точность выставляется по умолчанию равной 6. При этом ширина поля увеличивается. Форматирование: выравнивание по правому краю, знак указывать только для отрицательных чисел, точность равна 6, ширина поля равна 8.
4. Указана только точность, ширина поля подстраивается автоматически. Форматирование: выравнивание по правому краю, знак указывать только для отрицательных чисел, точность равна 1, ширина поля равна 3.
5. Форматирование: выравнивание по правому краю, знак указывать только для отрицательных чисел, точность равна 5, ширина поля равна 7.
6. Флаг «плюс» указывает на то, что следует выводить знак числа. Форматирование: выравнивание по правому краю, знак указывать для всех чисел, точность равна 2, ширина поля равна 6
7. Флаг «дефис» устанавливает выравнивание по левому краю. Если точность равна нулю, то число округлится до целого. Форматирование: выравнивание по левому краю, знак указывать только для отрицательных чисел, точность равна 0, ширина поля равна 7.
8. Флаг «ноль» выводит нули вместо лишних пробелов. Флаг «плюс» указывает знак числа. Форматирование: выравнивание по правому краю,

знак указывать для всех чисел, точность равна 2, ширина поля равна 7, между знаком числа и числом выводим нули.

9. Флаги «плюс», «дефис», «ноль». При сочетании флагов «дефис» и «ноль» флаг «ноль» не учитывается, так как при выравнивании по левому краю нет лишних пробелов в начале. Форматирование: выравнивание по левому краю, знак указывать для всех чисел, точность равна 2, ширина поля равна 7.

Если мы заранее не знаем, какая нам нужна точность и/или ширина поля, то мы можем указать эти величины в качестве параметров, а в строке форматирования использовать знаки «*».

Схема подстановки при использовании *



Примеры использования знака «*»

printf ("%*d", 5, 123);					1	2	3			
printf ("%*.*lf", 7, 2, 2.157);						2	.	1	6	
printf ("%4.*lf", 1, 2.157);			2	.	2					
printf ("%*.2lf", 6, 2.157);				2	.	1	6			

Примеры вывода символов и строк.

printf ("%5c", 'h');						h									
printf ("%-3c", 'k');		k													
printf ("%s", "hello!");		h	e	l	l	o	!								
printf ("%s", "I like cookies");		I		l	i	k	e		c	o	o	k	i	e	s

Вывод нескольких значений разных типов. Сложное форматирование.

Если вы хотите вывести не одно, а сразу несколько значений, то это нужно отразить и в строке форматирования, и в списке параметров. Например,

```
printf("%d %lf %d %c", 45, 95.1, -72, 'f');
```

выведет нам в строку через пробел: **45 95.100000 -72 f**

Если же мы не поставим пробелы в строке форматирования

```
printf("%d%lf%d%c", 45, 95.1, -72, 'f');
```

то увидим: **4595.100000-72f**

Если вы не ставите пробелы в строке форматирования, то и на экране их тоже не будет.

Во многих случаях нам нужно не просто увидеть цифры или буквы на экране, а знать, что каждая из них значит. Чаще всего на экран выводят не абстрактные числа, а переменные, которые содержат в себе результаты вычислений.

```
x = 45;
y = 95.1;
z = -72;
printf("x = %d  y = %.1lf  z = %d", x, y, z );
```

на экране появится: **x = 45 y = 95.1 z = -72**

```
int x = 27, y = 10;
double z = x / y;
printf("результат вычисления x/y = %lf", z);
```

на экране появится: **результат вычисления x/y = 2.000000**

Можно вывести результат вычисления более подробно:

```
int x = 27, y = 10;
double z = x / y;
printf("результат вычисления %d/%d = %.2lf", x, y, z);
```

тогда вы увидите: **результат вычисления 27/10 = 2.00**

В случае если нам нужно вывести информацию на нескольких строчках, это можно сделать с помощью управляющих последовательностей и экранирования.

Экранирование (\ ) — это операция, позволяющая заменять в тексте управляющие символы на соответствующие текстовые подстановки.

Экранирование (обратный слеш)	
управляющие символы	**значение**
\n	новая строка
\t	табуляция
\'	одинарная кавычка
\"	двойная кавычка
\\	обратный слеш
%%	знак процента

Управляющие последовательности не требуется отделять пробелами. Они пишутся слитно с текстом.

```
printf("Строка1\nСтрока2\nСтрока3");
```

Строка1

Строка2

Строка3

```
printf("Это обычная строка\n\tA это красная строка");
```

Это обычная строка

А это красная строка

```
printf("Одинарная кавычка \' используется редко.");
```

Одинарная кавычка ' используется очень редко.

```
printf("Прямая речь заключается в двойные кавычки \" \".");
```

Прямая речь заключается в двойные кавычки " ".

```
printf("Ставка по кредитам не более %d%% годовых.", 5);
```

Ставка по кредитам не более 5% годовых.

```
printf("x = %d\ny = %.2lf\n\tz = %c", 5, 6.2, 'j');
```

x = 5

y = 6.20

z = j

Функция ввода с клавиатуры `scanf()`

Функция `scanf()` считывает с клавиатуры символы, строки, числа и записывает их в адрес указанных переменных. Синтаксис функции очень похож и даже практически идентичен функции `printf()`

```
scanf("строка форматирования", адреса переменных);
```

Строка `scanf("%d", &x);` означает, что мы считываем с клавиатуры целое число и записываем его в адрес `(&)` переменной `x`. Фактически переменная `x` станет равной числу, которое мы введем с клавиатуры.

`&(амперсанд)` — это операция взятия адреса.

```
scanf("%lf", &y);     // дробное значение для переменной y
scanf("%d %lf", &a, &b); // целое a и дробное b через пробел
scanf("%c %c", &str1, &str2); // два символа через пробел
```

Так же можно использовать сложное форматирование:

```
scanf("часы = %d минуты = %d", &hour, &min);
```

здесь мы должны учесть форматирование, то есть мы должны напечатать **«часы = 12 минуты = 34»**, соблюдая все слова и пробелы. Таким же образом можно вводить даты:

```
scanf("%d.%d.%d", &day, &month, &year);
// печатаем «15.05.2023»
```

```
scanf("мимо пробегали %d розовых слоников", &elephants);
```

здесь тоже печатаем фразу целиком: **«мимо пробегали 8 розовых слоников».** Проще говоря, как написано в строке форматирования, так и нужно вводить данные с клавиатуры.

Операторы

Операторы ветвления. Операторы условного перехода

В языке Си существуют три оператора, обеспечивающие выполнения ветвления в программе: ? : , **if** и **switch**. При определенных обстоятельствах оператор ? : является альтернативой оператора **if**.

Оператор ?:

Оператор ?: Называют тернарным оператором (от лат. ternarius — «тройной»)

Общий вид операции:

```
(Условие) ? Оператор1 : Оператор2;
```

Если условие выполняется, то исполнятся Оператор1, если нет — то Оператор2.

Пример:

```
int x, y, max;
scanf("%d %d", &x, &y);
max = (x > y) ? x : y;
printf("Наибольшее число равно %d", max);
```

Тернарную операцию можно «прочитать» так: если `x` больше `y`, то переменная `max` равна `x`, если нет, то переменная `max` равна `y`.

Тернарную операцию можно включать в другие операции, сокращая при этом количество кода.

```
int x, y;
scanf("%d %d", &x, &y);
printf("Наибольшее число = %d", (x > y) ? x : y);
```

Здесь переменная максимум не обязательна. Вычисление наибольшего из `x` и `y` происходит прямо в функции `printf()` без использования промежуточной переменной.

Тернарные операции так же могут быть вложенными. В качестве примера можно рассмотреть поиск наибольшего из трех различных чисел.

В две строки (нет вложенности):

```
int a, b, c, max;
scanf("%d %d %d", &a, &b, &c);
max = (a > b) ? a : b;
max = (max > c) ? max : c;
printf("Наибольшее число равно %d", max);
```

Сначала находим наибольшее из `a` и `b` и записываем его в `max`. Затем в `max` записываем максимальное из `max` и `c`. Таким образом находим максимальное из трех чисел. При этом можно уменьшить код и использовать вложенные операции:

В одну строку (вложенность):

```
int a, b, c, max;
scanf("%d %d %d", &a, &b, &c);
max = (((a > b) ? a : b) > c) ? ((a > b) ? a : b) : c;
printf("Наибольшее число равно %d",  max);
```

Здесь так же можно заключить тернарную операцию сразу же в функцию `printf()`

```
printf("%d",(((a > b) ? a : b) > c) ? ((a > b) ? a : b) : c);
```

Такой вид записи условия возможен, но представляет значительную трудность для понимания. В случае сложных условий следует использовать оператор условия **`if-else`**

Оператор `if`

Общая (полная) форма оператора `if` следующая:

```
if(выражение/условие)
     оператор;

 else
     оператор;
```

```
if(выражение/условие) {
     блок операторов;
}
else {
     блок операторов;
}
```

Если выражение в скобках истинно (любое значение, отличное от нуля), то выполняется оператор или блок операторов, следующий за `if`. В противном случае выполняется оператор (или блок операторов), следующий за `else`. Необходимо помнить, что выполняется либо оператор, связанный с `if`, либо связанный с `else`, но оба — никогда!

Условный оператор может иметь неполную форму, при которой `else` и последующий оператор (или блок операторов) отсутствуют.

```
    if (выражение/условие)
{
        оператор;
```

```
if   (выражение/условие)

     блок операторов;
}
```

Условное выражение должно иметь скалярный результат. Это значит, что результатом должно быть целое число, символ, указатель или число с плавающей точкой, но им не может быть массив или структура. В выражении/условии оператора `if` результат типа с плавающей точкой используется редко, потому что это существенно замедляет вычислительный процесс. Объясняется это тем, что для выполнения операций над переменными дробного типа необходимо выполнить больше команд процессора, чем для выполнения операций над целыми числами или символами. Кроме того, следует избегать сравнения различных типов данных. Оно может быть некорректным.

Пример 1.

Модуль целого числа.

```
int x;
printf("Введите x:");
scanf("%d", &x);
if (x < 0)
     x *= -1;
printf("модуль x = %d\n", x);
```

Если мы ввели с клавиатуры отрицательное число, то для вычисления модуля нужно умножить его на минус единицу. Если же число не отрицательно, то, ни на что умножать не надо, его модуль будет равен самому числу. Поэтому отсутствует часть **else**.

Пример 2.

Вычислить значение переменной с условием.

```
int x = 5, y = 7;
double z;
if ((x * y > 50)  ||  (x / y < 10))
     z = (x + y) * 10.4;
else
     z = (35 * x - 12 * y) * 0.5;
printf("z=%lf",  z);
```

Вычисление значения переменной z в зависимости от условий. Переменная вычисляется по одной из двух формул и результат вычисления выводится на экран. Если у вас не одно, а несколько условий, то каждое из них рекомендуется заключить в круглые скобки. При этом не забывайте об общих круглых скобках.

```
if ((условие1) || (условие2) && (условие3))
```

Пример 3.

Вычисление ширины квадрата.

```
int square = 621, length = 0, height = 0;
printf("Enter length");
scanf("%d", &length);
if (length > 0) {
```

```
        height = square / length;
        printf("height = %d\n", height);
    }
    else
        printf("ERROR: length is wrong!\n");
```

Вычисление ширины квадрата, если известна его площадь, а длина вводится с клавиатуры. Если длина положительная, то мы вычисляем ширину. Иначе мы выдаем сообщение о том, что длина квадрата введена неверно.

Здесь после **if** используются фигурные скобки, которые заключают в себе блок операторов. Блоком операторов можно считать любое количество операторов (больше двух), которые заключены в фигурные скобки и выполняются все вместе. Если после **if** или после **else** вам требуется выполнить несколько операторов, то это можно сделать заключив их в фигурные скобки. В противном случае (если фигурные скобки пропущены), после **if** или после **else** выполнится только один самый первый оператор.

Невнимательность в вопросе проставления фигурных скобок приводит к большому количеству ошибок. Если вы часто совершаете подобные ошибки или не уверены в себе, то рекомендуется всегда ставить фигурные скобки, даже когда у вас всего один оператор после **if** и/или после **else**. Это поможет избежать значительное количество ошибок.

Пример 4.

Наибольшее из трех чисел .

```
int a, b, c;
scanf("%d %d %d", &a, &b, &c);
if (a > b)
    if (a > c)
        printf("a – наибольшее число");
    else
        printf("c – наибольшее число");
else
    if (b > c)
        printf("b – наибольшее число");
    else
        printf("c – наибольшее число");
```

Операторами для **if** и **else** могут быть так же условные операторы. При этом вложенность может быть многократная.

Условный оператор множественного выбора (переключатель) `switch`.

Синтаксис оператора

```
switch (селектор){
     case значение1:
          блок операторов;
          break;
     case значение2:
          блок операторов;
          break;
     case значение3:
          блок операторов;
          break;
     case значение4:
          блок операторов;
          break;
     case значение5:
          блок операторов;
          break;
     default:
          блок операторов;
          break;
}
```

Селектор — это переменная или выражение, которое должно иметь целочисленный или символьный тип. Оператор **`switch`** имеет от двух и более ветвей исполнения. Каждая ветвь начинается с ключевого слова **`case`**, за ним следует значение селектора, при котором должна выполняться данная ветвь. Чтобы после завершения кода ветви произошел выход из оператора переключателя, используется специальная команда **`break`**. Если такой команды в ветви нет, после исполнения кода выбранной ветви начнется исполнение кода всех следующих за ней ветвей. Это может служить причиной труднообнаруживаемых ошибок (если случайно пропустить **`break`**, компилятор не выдаст ошибки, но программа будет выполняться неверно). Ветвь **`default`** исполняется тогда, когда среди прочих ветвей не нашлось ни одной подходящей.

```
printf("Введите оценку ученика");
scanf("%d", &mark);
switch(mark){
case 2:
     printf("Неуспевающий ученик");
     break;
case 3:
     printf("Слабый ученик");
     break;
case 4:
     printf("Хорошист");
     break;
case 5:
     printf("Отличник");
     break;
 default:
     printf("Неверная оценка ученика") ;
     break;
}
```

Выбор условного оператора

При написании программы выбор условного оператора следует делать, исходя из задач и сложности реализуемой программы. Если условие, которое нужно реализовать, достаточно мало, то стоит попробовать обратиться к тернарной операции. Если же ветвление вашей программы идет в зависимости от вполне конкретных вариаций переменной целого или символьного типа, то оперировать стоит условным оператором **`switch`**`()`. На практике чаще всего используется условный оператор **`if-else`**. Этот оператор можно использовать для всех случаев ветвления программ. Но не стоит забывать о других, под час более удобных и кратких условных операторах.