

Go. Уровень 1

# Базовые типы данных. Основы работы с подсистемой ввода-вывода операционной системы. Основы структурирования Go-приложений

---



# На этом уроке

1. Познакомимся с простыми (базовыми) типами данных в Go.
2. Познакомимся с понятием стандартных потоков ввода-вывода (stderr, stdin, stdout).
3. Попрактикуемся в написании Go-приложений, использующих стандартные потоки ввода-вывода.
4. Познакомимся с принципами структурирования Go-приложений.
5. Познакомимся с понятием структур данных и алгоритмов.

## Оглавление

[Понятие данных и их типов](#)

[Понятие переменной](#)

[Как поменять местами значения переменных](#)

[Понятие константы](#)

[Простые \(базовые\) типы данных в Go](#)

[Числовые типы](#)

[Целые числа](#)

[Руны](#)

[Тип данных byte](#)

[Тип данных uintptr](#)

[Числа с плавающей точкой](#)

[Комплексные числа](#)

[Длинные числа](#)

[Полезные функции и константы из библиотеки math](#)

[Строки](#)

[Булев тип](#)

[Стандартные потоки ввода-вывода \(stdin, stdout, stderr\)](#)

[Простейшие вычисления](#)

[Структуры данных и алгоритмы](#)

[Эффективность и оценка алгоритмов](#)

[Практическое задание](#)

[Дополнительные материалы](#)

# Понятие данных и их типов

Вы уже знакомы с понятием типа данных из других языков программирования. Например, в курсе по JavaScript вы успели поработать как с простыми типами данных — строками, числами и булевыми значениями — так и с более сложными типами, такими, как массивы.

На этом уроке мы повторим то, что уже знаем о типах данных, а также разберёмся, какие простые типы данных существуют в Go.

По сути всё программирование строится именно на работе с данными. **Данные (data)** — это информация, которую хранят и которой манипулируют программы. Например, при работе с пользователями данными могут быть имя или возраст. В зависимости от назначения данных мы будем выбирать для них разные **типы (data type)**: для имени возьмём тип данных «строка», а для возраста — «целое неотрицательное число».

Типы данных делятся на:

- **простые** — сюда входят числа, строки, символы, булевы данные;
- **сложные** (ещё говорят «составные» или «композиционные»), которые состоят из других типов данных. Например, массив чисел — это сложный тип данных.

Также типы данных можно условно разделить на **встроенные** и **«пользовательские»**. Встроенные типы данных — это те, что входят непосредственно в язык. «Пользовательские» типы данных определяются в библиотеках, в том числе в стандартной библиотеке Go и пользовательских приложениях.

Забегая вперёд, заметим, что чаще всего в реальных приложениях мы будем сталкиваться именно с «пользовательскими» типами данных, но «под капотом» у них всегда будут встроенные. Изучение Go мы начнём именно со встроенных типов — простых и составных. Им мы посвятим этот урок и несколько ближайших.

Как правило, в рамках работы приложения мы производим **операции (operations)** над данными. В зависимости от типов данных операции могут быть разными. Например, для чисел мы можем использовать операцию возведения в степень, а для строк — операцию объединения (конкатенации).

Для выполнения операций мы будем использовать **операторы (operator)**. Например, в записи `"2 + 3"` оператор — `+`. Список всех операторов Go можно посмотреть на [сайте Golang](#). Мы будем знакомиться с ними постепенно.

Запись `"2 + 3"` называется **выражением (expression)**. Составляющие выражения, над которыми производится операция (то есть в данном случае `"2"` и `"3"`), называются **операндами (operand)**. Мы будем использовать эти понятия в курсе, да и в обычной программистской жизни они тоже пригодятся.

# Понятие переменной

Чтобы манипулировать данными, нам понадобится сохранять результаты операций. В рамках работы приложений для хранения данных используется оперативная память компьютера. У каждой ячейки памяти есть свой адрес, но такая адресация по номерам и прямое взаимодействие с памятью для доступа к данным были бы неудобными. Поэтому при разработке на языках программирования мы пользуемся более абстрактным понятием, которое называется «**переменная**». Это синоним адреса ячейки памяти. В книгах по программированию для начинающих переменную часто описывают как ящик, в который можно поместить одно значение конкретного типа. Чтобы не перепутать ящики, мы даём им уникальные имена. Иначе говоря, **любая переменная в Go характеризуется двумя параметрами: типом и именем**.

Чтобы начать работу с переменной, её сначала надо **объявить**, то есть задать ей имя и свойства. Общий вид объявления переменной в Go выглядит так:

```
var name type = value
```

Здесь `var` — ключевое слово, которое используется для объявления переменной, `name` — имя переменной, `type` — её тип и `value` — значение или выражение. Например, вот так можно объявить переменную `name` целого типа `int` и присвоить ей значение 5:

```
var name int = 5
```

Если при объявлении переменной опустить значение, в Go ей будет присвоено **нулевое значение**:

```
var name int
```

Для каждого типа данных нулевое значение своё. Например, для целых чисел это 0, для строк — пустая строка, для булева типа — `false`. При знакомстве с разными типами данных мы обязательно будем упоминать и их нулевые значения.

В Go также можно объявить в одной строке сразу несколько переменных:

```
var i, j, k int
var s, d, b = "Hello", 1.5, true
```

Помимо значений переменным можно присваивать и результаты вычисления выражений:

```
var a int = 5
var b int = a + 8

// Записываем в переменную дескриптор файла, имя которого задано в filename:
var filename := "./test.txt"
var f = os.Open(filename)
```

Также в Go можно использовать краткое объявление переменных:

```
name := value
```

В таком случае тип переменной **name** определяется по типу выражения **value**.

**Важно!** Go — это язык со строгой статической типизацией. Это значит, что в момент объявления переменной мы задаём ей конкретный тип и данные ей можно присваивать только этого типа.

## Как поменять местами значения переменных

На собеседованиях часто предлагают задачу о замене местами значений переменных. Например, у нас есть переменная **i** со значением 5 и переменная **j** со значением 8. Самый очевидный способ поменять местами две переменные — завести третью:

```
i := 5
j := 8

var k int
k = i
i = j
j = k
```

Можно обойтись и двумя переменными:

```
i := 5
j := 8

i = i + j
j = i - j
i = i - j
```

Более того, в случае с Go замену переменных можно сделать в одну строчку:

```
i := 5
j := 8

i, j = j, i
```

## Понятие константы

Помимо переменных нам также часто придётся иметь дело с константами. **Константы** представляют собой неизменяемые значения и создаются в момент компиляции. Константы можно определить только для чисел, рун, строк и булева типа данных. Для определения константы нам понадобится ключевое слово `const`:

```
const E = 2.71828182845904523536028747135266249775724709369995957496696763
```

Как и переменные, константы могут быть операндами в выражениях. Однако константу нельзя переопределить, и при попытке присвоить константе новое значение произойдёт ошибка компиляции.

Также при работе с константами полезным может быть генератор `iota`. При задании последовательности констант мы можем в объявлении указать `iota`, и тогда константы будут сгенерированы с нуля, и каждая следующая константа будет на единицу больше предыдущей. Работу с генератором констант проще всего понять на примерах из пакетов `time` и `net`.

В пакете `time`:

```
const (
    Sunday Weekday = iota
    Monday
    Tuesday
    Wednesday
    Thursday
    Friday
    Saturday
)
```

В пакете `net`:

```
const (
    FlagUp          Flags = 1 << iota // interface is up
    FlagBroadcast   // interface supports broadcast access
    capability
    FlagLoopback    // interface is a loopback interface
    FlagPointToPoint // interface belongs to a point-to-point link
    FlagMulticast   // interface supports multicast access
    capability
)
```

)

## Простые (базовые) типы данных в Go

Типы данных различаются по **размеру**, то есть занимают разное количество ячеек памяти. Вся память — это набор бит, каждый из которых представляет или значение 0, или значение 1. Таким образом, один бит может дать нам информацию о двух значениях — 0 или 1. Если мы возьмём два соседних бита, их возможными комбинациями будут 00, 01, 10 и 11, то есть два бита позволяют нам закодировать четыре разных значения. Три бита дадут уже восемь значений: 000, 001, 010, ..., 110, 111. Если мы возьмём восемь бит, то получим уже 256 значений, а для тридцати двух бит значений будет 4294967296.

Чтобы «разложить» обычное десятичное число в памяти, нам нужно определить его репрезентацию в виде битов или, иными словами, перевести число из десятичной в двоичную систему счисления. В этом курсе мы не будем останавливаться на системах счисления, но рекомендуем вам самостоятельно изучить эту тему. Можно начать со [статьи в «Википедии»](#) (обратите внимание на ссылки в конце статьи) или любой книги, рассказывающей об устройстве компьютера. Такие книги есть даже [в формате манги](#).

Знакомство с типами данных в Go мы начнём с простых типов, их ещё называют базовыми или фундаментальными. Такие типы данных включают в себя числа, строки и булев тип данных.

# Числовые типы

Числовые типы делятся на целые числа, числа с плавающей точкой и комплексные числа.

## Целые числа

Для целых чисел в Go представлена **знаковая** и **беззнаковая** арифметика. Если мы предполагаем, что в переменной будут храниться как положительные, так и отрицательные числа, нам понадобится знаковый тип данных. Для неотрицательных чисел будет достаточно беззнакового.

Разберёмся, зачем нужны знаковые и беззнаковые типы. Пусть у нас есть 8 бит памяти. Мы знаем, что в восьми битах можно представить 256 значений. Если мы работаем только с неотрицательными числами, мы сможем сохранить в этих восьми битах любое число от 0 до 255. Если мы хотим хранить как неотрицательные, так и отрицательные числа, 256 различных значений нужно начать с -128, пройти 0 и закончить на 127. Если же мы хотим сохранить число 256 в знаковом типе данных, нам понадобится ещё один бит. Таким образом, использование беззнаковых типов (когда это возможно) позволяет нам сэкономить пространство в памяти.

В Go есть четыре знаковых типа данных для целых чисел: `int8`, `int16`, `int32` и `int64`. Число в названии типа указывает на количество бит, резервируемых для хранения значений. Для хранения беззнаковых целых чисел тоже есть четыре типа: `uint8`, `uint16`, `uint32` и `uint64`. Буква `u` в этой записи означает `unsigned` (беззнаковый).

Кроме того, есть типы `int` и `uint`. Их размер может определяться компилятором и архитектурой процессора. Например, в случае AMD64 и стандартного компилятора Go 1.15 эти типы будут соответствовать `int64` и `uint64`. На другом компьютере или при использовании другого компилятора этим типам могут соответствовать `int32` и `uint32`. Таким образом, при использовании типов `int` и `uint` нельзя гарантировать диапазон значений, которые можно хранить в этих типах. Часто при определении переменных для целых чисел рекомендуется явно указывать число байтов, выбрав конкретный тип — `int8`, `int16`, `int32`, `int64` — или их беззнаковые аналоги.

**Нулевое значение** для любого целого — значение 0.

## Руны

Иногда целочисленные типы используются для хранения символов. Так, в Go есть тип данных `rune`, это синоним типа `int32`. Значения этого типа должны представлять собой [символы Unicode](#).

## Тип данных byte

Ещё один похожий тип данных — `byte`. Это синоним для типа `uint8`.



## Тип данных uintptr

Ещё один целочисленный тип данных — `uintptr` — понадобится нам при работе с указателями.

## Числа с плавающей точкой

В Go для чисел с плавающей точкой есть два типа данных: `float32` и `float64`. Арифметика для чисел с плавающей точкой гораздо сложнее целочисленной, она описывается [специальным стандартом](#).

**Важно!** Поскольку числа с плавающей точкой не гарантируют точность, никогда не храните данные, связанные с деньгами (цены, расчёты, баланс), в виде чисел с плавающей точкой.

В случае работы с Go рекомендуется хранить такие данные в виде целых (например, `int64`). Для этого необходимо перевести цену в минимальные возможные единицы. Например, если товар стоит 32 рубля 10 копеек, его цену мы переведём в копейки — 3210 — и сохраним это число в `int64`. Вся арифметика, связанная с деньгами, также должна вестись именно с целыми. Если для удобства пользователей мы хотим отображать цену в рублях и копейках, действие по приведению цены в рубли и копейки должно выполняться последним в цепочке арифметических вычислений.

В качестве примера потери точности обратимся к [случаю](#), демонстрирующему перевод числа из `float32` в `float64`:

```
package main

import (
    "fmt"
)

func main() {
    var a float32 = 359.9
    fmt.Println(a) // 359.9

    var b float64 = float64(a)
    fmt.Println(b) // 359.8999938964844
}
```

Другой интересный [пример](#):

```
package main

import (
    "fmt"
)

func main() {
```

```
var a float32 = 358.99999
var b float32 = 359.00001

fmt.Println(b - a) // 0
}
```

## Комплексные числа

Комплексные числа — не самый популярный тип данных. Тем не менее в Go этот тип встроенный, и есть два размера для их хранения — `complex64` и `complex128`. Соответственно, первый размер для хранения действительной и мнимой частей использует `float32`, а второй — `float64`.

Для работы с комплексными числами нам пригодятся три функции:

- `complex` для создания комплексного числа;
- `real` для взятия действительной части;
- `imag` для взятия мнимой части.

Также для работы с мнимой частью можно использовать литерал `i`. Например, мнимая часть может быть записана как `1i`, `2i`, `3i`.

Например:

```
package main

import "fmt"

func main() {
    x, y := complex(1, 2), complex(3, 4)

    fmt.Println(x, y)           // (1+2i) (3+4i)
    fmt.Println(real(x), imag(x)) // 1 2
    fmt.Println(x * y)          // (-5+10i)

    fmt.Println(1i)             // (0+1i)
    fmt.Println(1i * 1i)        // (-1+0i)
}
```

## Длинные числа

Изредка бывает, что даже длины `int64` или `float64` не хватает. В таких случаях на помощь придёт арифметика для длинных чисел. В Go такая арифметика реализована в библиотеке [math/big](#). Чаще

всего она нужна только для решения очень специфических задач, например олимпиадных. Обратите внимание: длинные числа с плавающей точкой по-прежнему не защищены от потери точности.

## Полезные функции и константы из библиотеки `math`

Для базовой арифметики мы в основном будем пользоваться операторами:

+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Взятие остатка от целого

Однако для более продвинутой работы с числами нам пригодятся функции и константы из библиотеки [math](#). Эта библиотека содержит тригонометрические функции, логарифмы и экспоненциальные функции, функции для округления чисел и другие полезные алгебраические преобразования. Среди констант можно найти основание натурального логарифма  $e$ , число  $\pi$ , квадратный корень числа 2 и другие полезные значения. Также эта библиотека содержит минимальные и максимальные значения чисел с плавающей точкой и максимальные значения целых.

## Строки

Мы уже немного говорили о битах и байтах. Строки в Go представляют собой неизменяемые последовательности байтов. Теоретически в этих байтах может храниться всё что угодно, но на практике в строках обычно всё же хранят человекочитаемую информацию. Строки интерпретируют как последовательность [Unicode-символов](#) в кодировке UTF-8. Это значительно упрощает работу со строками. Важно, что один символ в Unicode может занимать сразу несколько байтов.

Вернуть информацию о количестве байтов, занимаемых строкой, можно с помощью функции `len`. Обратиться к конкретному байту строки можно по его индексу, то есть порядковому номеру. Индексы начинаются с нуля.

Строки можно сравнивать и конкатенировать (соединять). Более сложная работа со строками возможна с помощью функций пакета [strings](#).

Рассмотрим [пример работы со строками](#):

```
package main

import (
```

```

    "fmt"
)

func main() {
    s := "Hello, " + "World!"
    fmt.Println(s)
    fmt.Println(len(s))

    s = "abcd"
    fmt.Println(len(s)) // Длина 4
    fmt.Println(s[0], s[1], s[2], s[3]) // 97 98 99 100

    s = "🐼"
    fmt.Println(len(s)) // Длина тоже 4
    fmt.Println(s[0], s[1], s[2], s[3]) // 240 159 164 147
}

```

В этом примере и последовательность символов "abcd", и эмодзи занимают по 4 байта. В случае обращения по индексу мы получаем информацию, записанную в байтах, а не отдельные символы.

## Булев тип

Булев тип в Go обозначается ключевым словом `bool` и имеет только два возможных значения: `true` (истина) или `false` (ложь). Работа со значениями булевого типа строится по правилам [булевой алгебры](#). Если вы ещё не изучали булеву алгебру, мы рекомендуем уделить этому время. Например, можно почитать книгу «[Дискретная математика для программистов](#)», там же вы встретите и другие дисциплины классического Computer Science.

Пока для нас важно, что булев тип используется в условиях конструкций `if` и `for`, а также в сравнениях.

Рассмотрим [пример](#):

```

package main

import (
    "fmt"
)

func main() {
    a := 5
    b := 7

    fmt.Println(a > b, a < b, a == b) // false true false

    if a < b {

```

```

        fmt.Println("Hello!")
    }
}

```

Здесь мы не определяем булев тип явно, но работаем с ним в рамках операций сравнения и конструкции `if`.

При использовании булевых значений для нас важны операторы `&&` (логическое И) и `||` (логическое ИЛИ). Эти операторы работают по правилам булевой алгебры.

Пусть `a` и `b` — какие-то булевы значения. Результатом операции `&&` будет `true`, только если и `a`, и `b` были оба `true`. Результатом операции `||` будет `false`, только если и `a`, и `b` были `false`.

Эти правила удобно представить в виде таблиц:

<code>&amp;&amp;</code>	<code>true</code>	<code>false</code>
<code>true</code>	true	false
<code>false</code>	false	false

<code>  </code>	<code>true</code>	<code>false</code>
<code>true</code>	true	true
<code>false</code>	true	false

Также есть унарный оператор логического отрицания `!`. Он меняет значение переменной на противоположное:

```

package main

import (
    "fmt"
)

func main() {
    a := true
    fmt.Println(!a) // false
    fmt.Println(!a) // true
}

```

Обратите внимание на приоритет операций: в выражении сначала выполняется операция `!`, затем `&&`, и только после этого `||`. Например:

```
!true || true && false // (!true) || (true && false)
```

## Стандартные потоки ввода-вывода (stdin, stdout, stderr)

Мы познакомились с основными типами данных и готовы начать писать самые первые программы. Чтобы приложения были интерактивными, понадобится источник входных данных. В самом простом случае таким источником может стать **подсистема ввода-вывода операционной системы**. Она отвечает за **обмен данными** между приложениями, пользователями и устройствами. Даже в самых первых приложениях нам не обойтись без обмена данными: уже в простейших консольных утилитах нужно выводить информацию на экран, считывать конфигурацию, запрашивать информацию от пользователя, выводить сообщения об ошибках и отладочную информацию.

Вспомните функцию `fmt.Println` из самого первого примера `"Hello, World!"`. Как бы вы описали, что она делает? Например, можно сказать, что она выводит сообщение на экран, а значит, взаимодействует с системой ввода-вывода. Если нам нужно считать данные с клавиатуры, мы можем воспользоваться «обратной» функцией `fmt.Scanln`.

Разберёмся, что же происходит «под капотом», когда мы используем эти функции. На самом деле для взаимодействия с экраном и клавиатурой мы используем **потоки ввода-вывода**. В английских источниках их можно встретить под названиями `in-out` или `I/O streams`. Именно они служат путями чтения информации из таких источников, как клавиатура, или записи информации в такие приёмники, как экран ноутбука. Кстати, если вы уже работали с утилитами командной строки Linux, вы знаете, что многие из них тоже активно взаимодействуют с потоками ввода-вывода не только для чтения данных непосредственно от пользователя, но и для передачи данных друг другу.

В таких случаях, когда мы говорим о чтении данных в программу, мы имеем дело со стандартным потоком ввода (`stdin`). Когда мы выводим данные из программы, мы работаем со стандартным потоком вывода (`stdout`). Например, если мы посмотрим на код функции `fmt.Scanln`, мы как раз увидим, что она работает с `stdin`, а функция `fmt.Println` — с `stdout`.

Ещё один стандартный поток — стандартный поток ошибок (`stderr`). Он похож на `stdout`, но его принято использовать для вывода информации об ошибках и отладочной информации. Снова вернёмся к примеру с утилитами командной строки Linux. Как правило, если при вызове утилиты происходит ошибка, её вывод как раз производится в поток `stderr`. Возможно, сейчас разница между `stdout` и `stderr` не так очевидна, но на практике она очень полезна. Например, если мы хотим как-то специально логировать ошибки, мы можем перенаправить поток `stderr` в нужный нам приёмник, при этом все важные для пользователя сообщения по-прежнему останутся в `stdout` и будут показаны на экране.

Для работы со стандартными потоками из Go также можно воспользоваться тремя переменными из пакета `os`: `os.Stdin`, `os.Stdout` и `os.Stderr`.

Например, вот такие вызовы будут эквивалентны:

```
fmt.Println("Hello, world!")
fmt.Fprintln(os.Stdout, "Hello, world!")
```

И аналогично в пару к `fmt.Scanln` идёт `fmt.Fscanln`, первый аргумент в котором задан как `os.Stdin`.

Ну и, как вы уже догадались, для вывода сообщения в стандартный поток ошибок мы снова можем воспользоваться функцией `fmt.Fprintln`:

```
fmt.Fprintln(os.Stderr, "Hello, world!")
```

Мы подробнее изучим работу со стандартными потоками ввода-вывода в дальнейших уроках этого и следующих модулей. А пока попробуем написать приложение, запрашивающее пользовательский ввод с клавиатуры.

## Простейшие вычисления

Чтобы было интереснее, в следующем уроке мы напишем не просто абстрактное приложение, а самый настоящий калькулятор. В этом уроке начнём с простого варианта: пусть пользователь вводит два числа, а мы их будем складывать:

```
package main

import "fmt"

func main() {
    var a, b float32

    fmt.Print("Введите первое число: ")
    fmt.Scanln(&a)

    fmt.Print("Введите второе число: ")
    fmt.Scanln(&b)

    fmt.Printf("Сумма чисел: %f\n", a+b)
}
```

Для того, чтобы дать пользователю возможность выбирать разные операции, нам понадобятся управляющие конструкции. В следующем уроке мы познакомимся с ними и сделаем настоящий калькулятор.

Пока продолжим практиковаться с простыми операциями.

## Структуры данных и алгоритмы

Для полного погружения в Computer Science помимо практики на Go нам также необходимо познакомиться с некоторыми фундаментальными аспектами программирования. В первую очередь со структурами данных и алгоритмами.

**Алгоритм** — это вычислительная процедура, **на вход** которой подаётся набор данных, и в качестве результата выполнения которой **на выходе** мы также имеем набор данных. При этом в рамках алгоритма выполняется последовательность шагов, преобразующая входные данные в выходные. Также помимо алгоритма нам важна **задача**, которую алгоритм должен решить. Даже когда мы говорим о простейших операциях, например, расчете периметра и площади, мы имеем дело с алгоритмами. Конечно, более интересными являются итеративные алгоритмы такие, как сортировка и поиск. С ними мы познакомимся в дальнейших уроках.

Ещё одно важное понятие в Computer Science — структуры данных. **Структура данных** — это способ организации и хранения данных, облегчающий доступ к этим данным и управление ими. При этом разные структуры данных ориентированы на решение разных задач. У каждой структуры данных есть свои ограничения в использовании, преимущества и недостатки.

В реальной практике разработки веб-приложений реализацию классических структур данных и алгоритмов не приходится часто писать с нуля. Однако их знание и понимание принципов работы критически важно, чтобы стать хорошим программистом, научиться выбирать оптимальные решения и избегать оверинжиниринга. Кроме того, часто работа с базовыми структурами данных и алгоритмами встречается в виде тестовых заданий и вопросов на собеседованиях. В таких случаях компании, как правило, хотят проверить эрудицию и серьёзность подхода кандидатов к вопросам программирования.

В рамках курса мы будем время от времени ссылаться на некоторые структуры данных и алгоритмы и предлагать их для самостоятельного изучения. Рекомендуем отнестись к этому вопросу серьёзно и разобрать всю предлагаемую теорию. В случае возникновения вопросов обязательно обращайтесь к преподавателю курса для разбора непонятных моментов непосредственно на занятиях.

## Эффективность и оценка алгоритмов

Алгоритмы, решающие одну и ту же задачу, могут различаться по эффективности. Как правило, при расчёте эффективности учитывают количество операций, которые нужно произвести для прохождения всего алгоритма.



На следующих уроках мы начнем разговор о сложности и оценке алгоритмов. Для более глубокого погружения в эту тему уже сейчас рекомендуем книгу «[Алгоритмы: построение и анализ](#)».

## Практическое задание

1. Напишите программу для вычисления площади прямоугольника. Длины сторон прямоугольника должны вводиться пользователем с клавиатуры.
2. Напишите программу, вычисляющую диаметр и длину окружности по заданной площади круга. Площадь круга должна вводиться пользователем с клавиатуры.
3. С клавиатуры вводится трехзначное число. Выведите цифры, соответствующие количеству сотен, десятков и единиц в этом числе.
4. Проверьте себя:
  - a. Вам должны быть знакомы следующие ключевые слова Go: `package`, `import`, `func`, `var`, `const`.
  - b. Вам должны быть знакомы следующие константы: `true`, `false`, `iota`, `nil`.
  - c. Вам должны быть знакомы следующие типы: `int`, `int8`, `int16`, `int32`, `int64`, `uint`, `uint8`, `uint16`, `uint32`, `uint64`, `float32`, `float64`, `complex128`, `complex64`, `bool`, `byte`, `rune`, `string`.
  - d. Вам должны быть знакомы функции `new`, `complex`, `real`, `imag`, `len`.

# Дополнительные материалы

## Для погружения в типы данных

1. [Книга «Язык программирования Go» \(Донован, Керниган\), главы 2 и 3.](#)
2. [Манга об устройстве компьютера.](#)
3. [О проблеме чисел с плавающей точкой.](#)
4. [Статья на русском о числах с плавающей точкой.](#)
5. [Очень продвинутое погружение в работу с памятью.](#)

## Для погружения в структуру приложения и управляющие конструкции

6. [Go tour: пакеты, переменные и функции.](#)
7. [Effective Go](#), главы Semicolons, Control structures, Initialization.

## Полезные пакеты

8. Встроенные типы: [пакет builtin](#).
9. Пакеты [fmt](#) и [io](#).

## Потоки ввода-вывода

10. [Разбираемся в Go: пакет io](#).
11. [Пакет os для работы с сущностями операционной системы](#) (см. Stdin, Stdout, Stderr).
12. [Продвинутое погружение в пакет io](#).
13. [Продвинутое погружение в стандартные потоки ввода-вывода](#).