

Оглавление

История.....	2
Установка Julia и обзор IDE	4
Язык Julia. Установка	4
Repl. Режимы работы в repl	6
IDE. Настройка IDE	8
IDE. Отладка.....	8
Переменные.....	12
Допустимые имена переменных.....	12
Комментарии	12
Типы данных	13
Тип Int	13
Тип Float.....	15
Математические операции.....	15
Операции сравнения	17
Тип Bool	19
Задания для самостоятельного решения.....	19
Символы и строки	21
Тип string.....	24
Задания для самостоятельного решения.....	25
Условные выражения в языке программирования Julia	26
Задачи для самостоятельного решения.....	29
Составные выражения на языке программирования Julia	30
Функции в Julia	32
Область действия переменных в функции	34
Задачи для самостоятельного решения.....	36
Массивы.....	37
Способы объявления массивов.....	37
Определённые операции над массивами	38
Выборки (slice).....	41
Задачи для самостоятельного решения.....	42
Кортежи	44
Словари.....	44
Задачи для самостоятельного решения.....	45
Циклы for, while	46
Цикл for.....	46
Область видимости переменных внутри циклов	47
Цикл while.....	49
Задачи для самостоятельного решения.....	50
Векторизованные операции.....	51
Задачи для самостоятельного решения.....	53
Работа с файлами в Julia.	54
Решения задач	56

Julia—это хорошо структурированный язык программирования с большим быстродействием, устраняющий классическую проблему выполнения анализа на одном языке и трансляции его результатов на второй с целью повышения производительности.

История

Julia—это относительно молодой язык программирования. Первоначальные проектные работы по языку Julia начались в Массачусетском технологическом институте (MIT) в августе 2009, и к февралю 2012 он стал общедоступным. Данный язык является в основном результатом работы трех разработчиков: Стефана Карпински, Джеффа Безансона и Вирала Шаха. Все трое, вместе с Аланом Эдельманом, до сих пор остаются активно преданными проекту Julia в MIT.

Язык Julia —это универсальный и мощный инструмент, который может помочь каждому исследователю практически в любой области научной или технической деятельности.

Язык программирования впервые выпустился в 2012 году в стенах Массачусетского технологического института (MIT)

В 2018 он дорос до версии 1.0

Язык в первую очередь ориентирован на научные вычислительные задачи, требующие высокой производительности, но и в качестве языка общего назначения он тоже применим.

Из преимуществ языка можно выделить:

- Простой синтаксис
- Высокую скорость разработки и исполняемого кода
- гибкую систему типов
- богатый набор библиотек под научные задачи
- удобные инструменты визуализации результатов

Язык Julia может быть применим в следующих областях:

- Научное программирование
- Машинное обучение и анализ данных
- Образование

Язык Julia —общие сведения (можно сравнить)

Если рассматривать Julia просто как способ записи алгоритмов, то в первом приближении кажется, что всё в нём практически совпадает с тем, что можно написать в MATLAB...

(a) <i>Julia</i>	(b) <i>MATLAB</i>	(c) <i>Python</i>
<pre># Это комментарий в Julia #= Многострочный комментарий =# if i <= N # Условие # Действия else # Альтернативные действия end while i <= N # Условие # Действия end for i = 1:N # Действия end</pre>	<pre>% Это комментарий в MATLAB %{ Многострочный комментарий }% if i <= N % Условие % Действия else % Альтернативные действия end while i <= N # Условие % Действия end for i = 1:N % Действия end</pre>	<pre># Это комментарий в Python # # Многострочный комментарий # if i <= N: # Условие # Действия else: # Альтернативные действия while i <= N: # Условие # Действия for i in range(N): # Действия</pre>

Примеры базового синтаксиса в языках Julia, MATLAB, Python.

Установка Julia и обзор IDE

Язык Julia. Установка

В данном разделе мы подробно остановимся на том, как установить и настроить язык Julia на разных операционных системах (ОС).

Итак, для начала заходим на сайт: <https://julialang.org/downloads/>

Download Julia

Star 40,913

Please star us on [GitHub](#). If you use Julia in your research, please [cite us](#). If possible, do consider [sponsoring us](#).

Current stable release: v1.8.2 (September 29, 2022)

Checksums for this release are available in both [MD5](#) and [SHA256](#) formats.

Windows [help]	64-bit (installer), 64-bit (portable)	32-bit (installer), 32-bit (portable)	
macOS x86 (Intel or Rosetta) [help]	64-bit (.dmg), 64-bit (.tar.gz)		
macOS ARM (M-series Processor) [help]	64-bit (.dmg), 64-bit (.tar.gz)		
Generic Linux on x86 [help]	64-bit (glibc) (GPG), 64-bit (musl) ^[1] (GPG)	32-bit (GPG)	
Generic Linux on ARM [help]	64-bit (AArch64) (GPG)		
Generic FreeBSD on x86 [help]	64-bit (GPG)		
Source	Tarball (GPG)	Tarball with dependencies (GPG)	GitHub

Далее вы скачиваете необходимый установщик для вашей ОС.

Если же вы работаете на Linux, то установка Julia также возможна как через встроенный менеджер пакетов, так и через snap пакет.

Для этого введите в терминале следующую команду:

`sudo apt install julia` – для установки из репозиториев дистрибутива (debian-based)

`sudo rpm install julia` – для установки из репозиториев дистрибутива RHEL

`sudo snap install julia` - для установки из snap-пакета

А при установке с официального сайта необходимо распаковать скачанный архив и создать символическую ссылку на папку куда вы его распаковали (в данном примере это папка `dir`, **ваше название может отличаться!**).

`sudo ln -s /dir/julia-1.8.3/bin/julia /usr/local/bin/Julia`

в рамках курса рекомендуется использовать последнюю версию Julia, скаченную с официального сайта.

Repl. Режимы работы в repl

Для проверки работоспособности Julia необходимо открыть терминал либо командную строку и ввести `julia`

Julia поставляется с полнофункциональной интерактивной командной строкой REPL (read-eval-print loop) : ввод пользователя считывается, исполняется, а результат исполнения выводится, после чего снова появляется REPL-подсказка и опять ожидается ответ пользователя. В такой среде пользователь может вводить выражения, которые среда тут же будет вычислять, а результат вычисления отображать пользователю. Выйти из этого интерактивного режима можно при помощи комбинации клавиш `Ctrl + D` или после ввода вызова функции `exit()`.

```
$ julia

      _
     _(_)
    (-) | (-) (-)
   _ _ | | _ _ _
  | | | | | | / _ ' |
  | | | | | | (- | |
 _/ | \ _ ' - | - | \ _ ' - |
| _/

Documentation: https://docs.julialang.org
Type "?" for help, "]?" for Pkg help.
Version 1.8.3 (2022-11-14)
Official https://julialang.org/ release

julia>
```

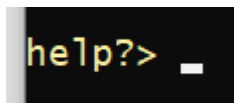
Давайте напишем нашу первую программу на Julia:

```
julia> println("Hello world!")
Hello world!
```

Здесь `println()` – функция которая печатает строку “hello world!” на экран и переносит каретку на новую строку, более подробно мы познакомимся с функциями дальше.

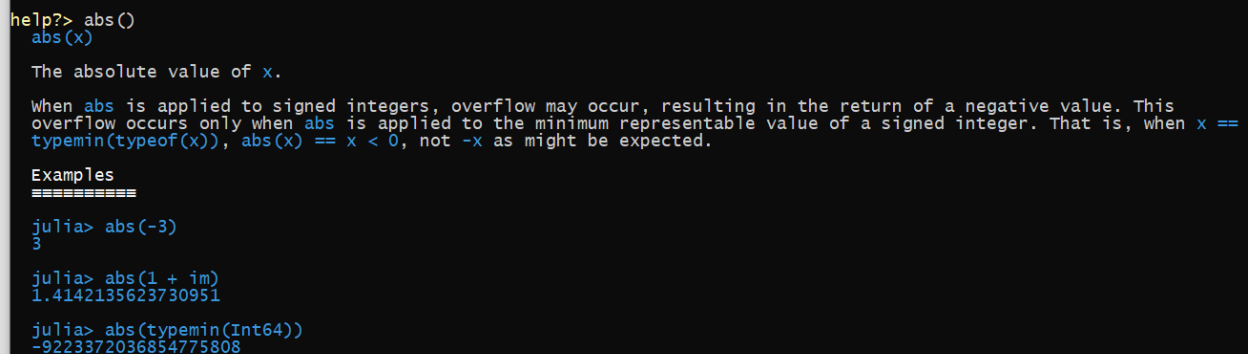
Также в языке Julia есть функция `print()` – она тоже печатает на экран, однако не переносит каретку на новую строку.

В REPL существует несколько режимов работы. На начальных этапах может понадобиться всего несколько режимов. Например, режим справки: `?` (для перехода в режим работы со справкой необходимо нажать вопросительный знак) после нажатия слово `julia>` заменится на `help?>` (подсказка `help?>` будет жёлтого цвета) или режим системной оболочки (подсказка `shell>` красного цвета; надо ввести точку с запятой `;`), поиск в истории введенных команд (подсказка белого цвета после `Ctrl + R`). Возврат к исполнению - нажатие клавиши `BackSpace` в самом начале строки (она же стирает всё введенное ранее).



В режиме работы со справкой пользователь может получить документацию по любым определенным в языке Julia операторам, функциям и ключевым словам.

Попробуйте вывести, например, справку о модуле числа: `?abs()`



```
help?> abs()
abs(x)

The absolute value of x.

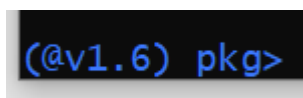
When abs is applied to signed integers, overflow may occur, resulting in the return of a negative value. This
overflow occurs only when abs is applied to the minimum representable value of a signed integer. That is, when x ==
typemin(typeof(x)), abs(x) == x < 0, not -x as might be expected.

Examples
=====
julia> abs(-3)
3

julia> abs(1 + im)
1.4142135623730951

julia> abs(typemin{Int64}())
-9223372036854775808
```

Так же не менее важным режимом работы REPL является режим работы с библиотеками (packages), его можно активировать введя на клавиатуре символ `]`



Библиотеки или же пакеты – это файлы, содержащие функции и структуры, помогающие в решении прикладных задач

С помощью команды `help` введенной в режиме работы с пакетами можно получить справку со всеми командами данного режима.

Давайте воспользуемся командой `add`.

Синтаксис у команды следующий: `add <имяПакета>`

Данная команда нужна чтобы устанавливать сторонние библиотеки из репозитория языка Julia.

Попробуйте установить пакет Plots при помощи следующей команды:
jadd Plots

IDE. Настройка IDE

Интегрированная среда разработки (IDE) – это программное приложение, которое помогает программистам эффективно разрабатывать программный код. Оно повышает производительность разработчиков, объединяя такие возможности, как редактирование, создание, тестирование и упаковка программного обеспечения в простом для использования приложении.

Для языка Julia существует ряд удобных редакторов и сред разработок, которые перечислены ниже:

- VS Code
- Geany
- Jupyter
- Pluto
- Vim

Вы можете в будущем использовать любым IDE, но в данном курсе мы остановимся на **VS Code**.

Перейдите по следующей ссылке и установите VS Code для своей платформы: <https://code.visualstudio.com/download>

Теперь необходимо установить расширение Julia в VS Code.

Запустите VS Code

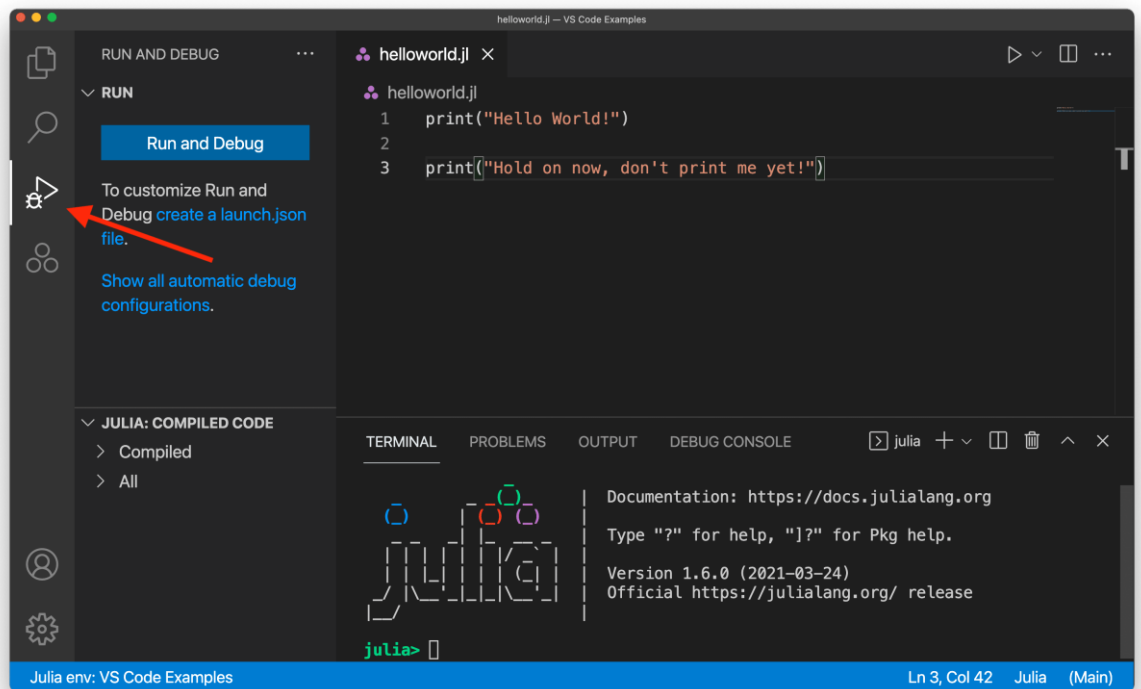
Внутри VS Code перейдите к просмотру расширений, нажав **View** верхней строке меню, а затем выбрав **Extensions**.

В **Extensions** найдите "julia" в поле поиска Marketplace, затем выберите расширение Julia (julialang.language-julia) и нажмите кнопку **Install**.

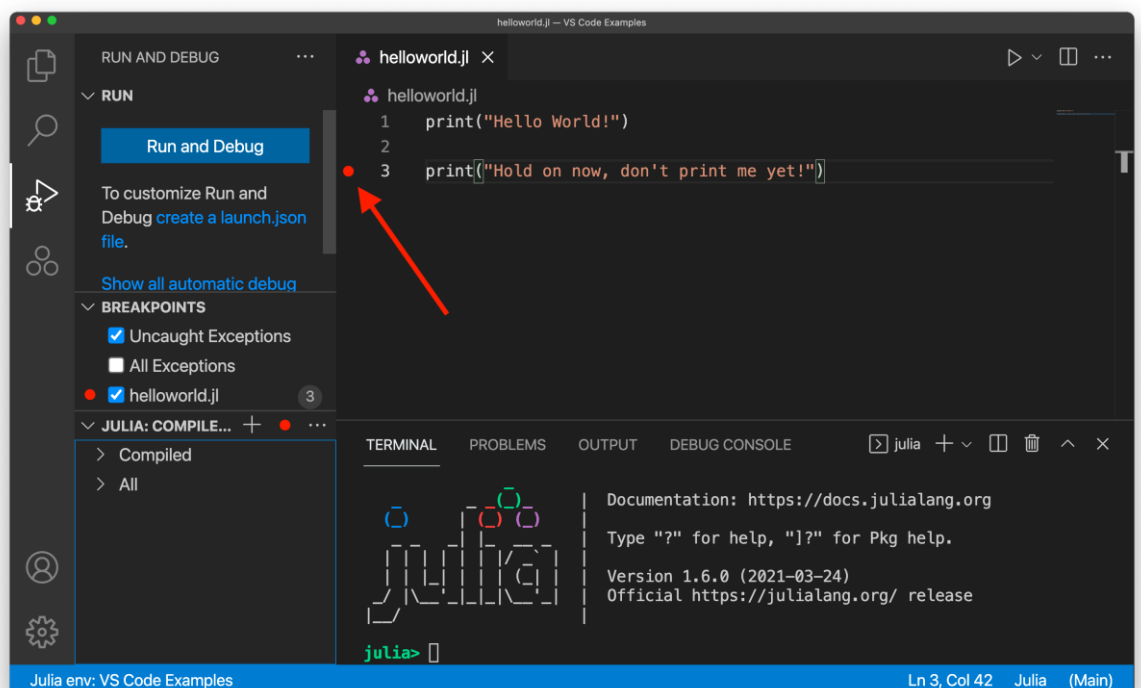
Перезапустите VS Code

IDE. Отладка

Отладку можно начать, открыв файл Julia, который требуется отладить. Затем выбрать **Run and Debug** на панели действий (как показано ниже):

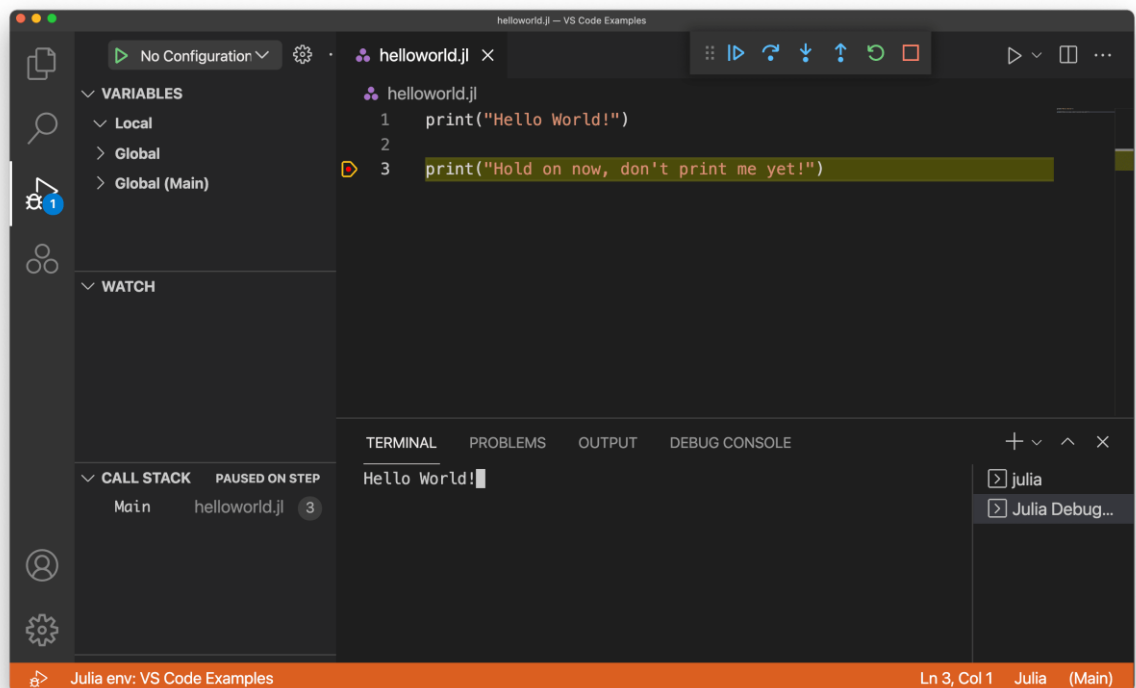


Затем вы можете добавить точку останова, щелкнув слева от номера строки:



После добавления точки останова (или любого другого типа конфигурации отладки) нажмите кнопку **Run and Debug** слева. Начало запуска может за-

нять несколько секунд. Затем вы должны увидеть выходные данные выполнения кода с конфигурацией отладки. В этом примере, поскольку мы добавили точку останова, вы увидите следующее:



После запуска сеанса отладки в верхней части редактора появится **панель инструментов Отладка**.



Continue: возобновить нормальное выполнение программы/скрипта (до следующей точки останова).

Step Over: выполняет следующую инструкцию в обычном пути выполнения программы. Однако в то время как Step Into будет входить в вызовы функций и выполнять их построчно, Step Over выполнит всю функцию без остановки и вернет вам управление после выполнения этой функции.

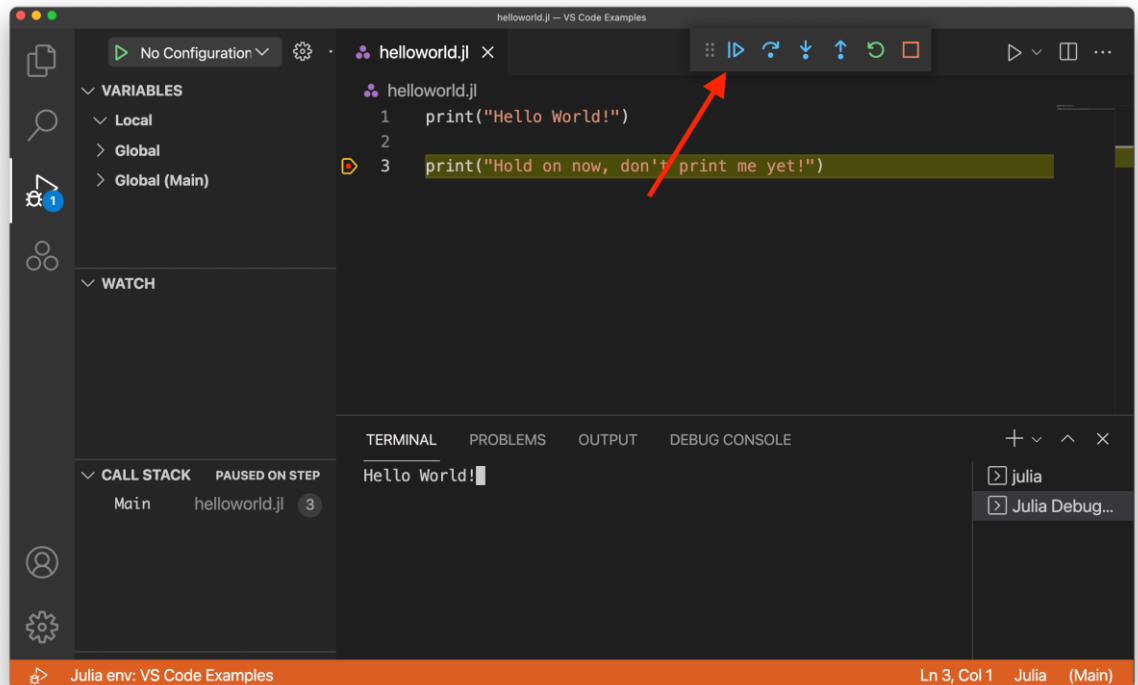
Step Into: команда выполняет следующую инструкцию в обычном пути выполнения программы, а затем приостанавливает выполнение программы, чтобы мы могли проверить состояние программы с помощью отладчика. Если выполняемый оператор содержит вызов функции, Step Into заставляет программу перескакивать в начало вызываемой функции, где она приостанавливается.

Step Out: в отличие от двух других пошаговых команд эта не просто выполняет следующую строку кода. Вместо этого он выполняет весь оставшийся код функции, которая выполняется в настоящее время, а затем возвращает управление вам, когда будет выполнен возврат из функции.

Restart: завершить текущее выполнение программы и снова начать отладку, используя текущую конфигурацию запуска.

Stop: завершить выполнение текущей программы.

Обратите внимание, что вторая команда печати еще не выполнена, и в терминале есть только текст из первой команды печати. Вы можете завершить выполнение программы, нажав кнопку **Continue**:



Домашнее задание: Установить язык Julia, установить VS Code

Переменные

Переменная в Julia – это имя, связанное со значением. Это полезно, когда вы хотите сохранить значение для последующего использования:

```
julia> x = 10
-----
10
julia> x + 1
-----
11
julia> x = 'wonderful day'
-----
wonderful day
```

Имена переменных могут быть сколь угодно длинными. Язык Julia чувствителен к регистру, т.е. переменные с именами `a` и `A` являются разными, но общепринято использовать только строчные буквы для имен переменных. Имена могут содержать почти все символы кодировки Unicode (в формате UTF-8), но не могут начинаться с цифры.

Допустимые имена переменных

Имена переменных в Julia:

- ✓ должны начинаться с подчеркивания, буквы (A-Z или a-z)
- ✓ могут содержать цифры (0-9) или восклицательный знак (!), но не должны начинаться с них
- ✓ можно использовать такие операторы, как `+`, `^` и т.д.

Комментарии

Можно использовать однострочные и многострочные (блочные) комментарии в тексте программы. В первом случае (одна строка) текст после символа `#` воспринимается компилятором как комментарий:

`#` Это комментарий, он не влияет на работу программы

Если комментарий достаточно велик и занимает несколько строк, его можно обозначить так:

`#=` это многострочный
комментарий `=#`

Типы данных

Тип Int

Тип	Со знаком?	Количество битов	Наименьшее значение	Наибольшее значение
<code>Int8</code>	✓	8	-2^7	$2^7 - 1$
<code>UInt8</code>		8	0	$2^8 - 1$
<code>Int16</code>	✓	16	-2^{15}	$2^{15} - 1$
<code>UInt16</code>		16	0	$2^{16} - 1$
<code>Int32</code>	✓	32	-2^{31}	$2^{31} - 1$
<code>UInt32</code>		32	0	$2^{32} - 1$
<code>Int64</code>	✓	64	-2^{63}	$2^{63} - 1$
<code>UInt64</code>		64	0	$2^{64} - 1$
<code>Int128</code>	✓	128	-2^{127}	$2^{127} - 1$
<code>UInt128</code>		128	0	$2^{128} - 1$

Целочисленные типы:

Давайте напишем программу, которая принимает на вход число с клавиатуры и выводит его на экран. Для этого воспользуемся функциями `readline()` и `parse()`

При вызове функции `readline()` Julia будет ожидать от пользователя ввод с клавиатуры, затем введенное значение преобразуется в строку (подробнее о строках будет рассказано далее), например

```
julia> a = readline()  
6  
"6"
```

Мы получили строковое значение равное символу '6', как получить из него число? Для этого поможет функция `parse()`

Синтаксис функции `parse()` выглядит следующим образом:

```
parse(<ТипДанных>, <ИсточникДанных>)
```

где ТипДанных - тип, в который мы хотим преобразовать наши данные.

ИсточникДанных – то, какие данные мы хотим преобразовать.

Например,

```
julia> b = parse{Int, a}

-----

6
```

По умолчанию тип целочисленной переменной определяется в зависимости от архитектуры используемой системы (32-битная или 64-битная). С помощью функции `typeof(<объект>)` можно получить информацию о типе данных указанного в скобках объекта, например:

```
julia> typeof(a)

-----

String

julia> typeof(b)

-----

Int64
```

В Julia также возможно указывать типы переменных явно, это может быть полезно для оптимизации программы и для уменьшения количества ошибок, связанных с типами данных. Указать тип переменной можно с помощью оператора `::`: например:

```
julia> x::Int64 = 6

-----

6
```

Здесь мы создали переменную `x` и явно указали, что в ней будет храниться значение типа `Int64`, равное 6.

Тип Float

Вещественные типы:

Тип	точность	Кол-во бит
Float16	Половинная	16
Float32	Одинарная	32
Float64	Двойная	64

Число типа Float16 – хранится в 16-битной ячейке памяти, а Float32 и Float64 соответственно в 32-битной и 64-битной. По умолчанию тип числа будет выбран в соответствии с разрядностью системы.

```
julia> 1.0
-----
```

```
1.0
julia> typeof(1.0)
-----
```

Float64

Математические операции

Для всех числовых типов поддерживаются следующие арифметические операторы:

Оператор	Описание
$-x$	Отрицание x
$x + y$	Сложение
$x - y$	Вычитание
$x * y$	Умножение

x / y	Деление
$x \div y$	Целочисленное деление
$x \setminus y$	Обратное деление, эквивалентно $y \setminus x$
$x ^ y$	Возведение x в степень y
$x \% y$	Остаток от деления x на y

Число, помещенное непосредственно перед идентификатором или круглыми скобками, например $2x$, или $2(x+y)$, обрабатывается как умножение, за исключением того, что имеет более высокий приоритет, чем другие двоичные операции.

```
julia> x = 10
```

```
-----
```

```
10
```

```
julia> 4x
```

```
-----
```

```
40
```

Также существуют комбинированные (составные) операторы, которые выглядят следующим образом:

Оператор	Эквивалентная запись
$x += y$	$x = x + y$
$x -= y$	$x = x - y$
$x *= y$	$x = x * y$
$x /= y$	$x = x / y$
$x \div y$	$x = x \div y$
$x \setminus y$	$x = x \setminus y$
$x ^ y$	$x = x ^ y$
$x \% y$	$x = x \% y$

¹ Чтобы ввести оператор целочисленного деления необходимо написать `\div` и нажать клавишу `tab`

Например,

```
julia> x = 1
```

1

```
julia> x += 3
```

4

```
julia> x
```

4

Операции сравнения

Стандартные операции² сравнения определены для всех числовых типов:

Оператор	Название
<code>==</code>	равенства
<code>!=, ≠</code>	неравенства
<code><</code>	меньше
<code><=, ≤</code>	меньше или равно
<code>></code>	больше
<code>>=, ≥</code>	больше или равно

Вот несколько простых примеров:

² Чтобы использовать `≤` введите `\le + tab`, для `≥` - `\ge + tab`, для `≠` - `\ne + tab`

```
julia> 1 == 1
```

```
-----
```

```
true
```

```
julia> 1 == 2
```

```
-----
```

```
false
```

```
julia> 1 != 2
```

```
-----
```

```
true
```

```
julia> 1 == 1.0
```

```
-----
```

```
true
```

```
julia> 1 < 2
```

```
-----
```

```
true
```

```
julia> 1.0 > 3
```

```
-----
```

```
false
```

```
julia> 1 >= 1.0
```

```
true
```

```
julia> -1 <= 1
```

```
true
```

```
julia> -1 <= -1
```

```
true
```

```
julia> -1 <= -2
```

```
false
```

```
julia> 3 < -0.5
```

```
false
```

Тип Bool

Логический тип данных имеет всего два возможных значения:

- true – истина
- false – ложь

логические операторы:

Оператор	Описание
! x	Отрицание
x && y	Логическое И
x y	Логическое ИЛИ

```
julia> x = true  
true
```

```
julia> !x  
false
```

```
julia> x && true  
true
```

```
julia> x || false  
true
```

Задания для самостоятельного решения

- 1) Вычислить значение $y = 2x^2 + x$ при $x = 3$.
- 2) Вычислить значение $y = x^3 - 5$ при $x = 2,5$.
- 3) Вычислить значение $z = xy - 4$ при $y = 0,5x - x^2$, $x = 3$.

- 4) Найти остаток от деления t на s , если $t = x + 566x^2$, $s = 27x^3$, $x = 4$.
- 5) Вывести результат сравнения $a > b$, $a < b$, $a == b$, если $a = 3.14x$, $b = 1.37x + x$ при $x = 5,2$.
- 6) Вывести результат сравнения $c != e$, $c \leq e$, $c \geq e$, если $c = 14 - 5e - x$, $e = x^4 + 5x$ при $x = 1,3$.

Символы и строки

Строковые типы данных в языке Julia делятся на:

- Символьные (Char)
- Строковые (String)

Значения типа Char представляют из себя один символ.

Элементы типа Char / String необходимо заключать в одинарные, либо же двойные кавычки.

```
julia> c = 'x' # присвоили в переменную c символ 'x'
```

```
julia> b = x # присвоили в b какой-то элемент x (не строковый)
```

Char символы стандартизированы вместе с отображением в целочисленные значения от 0 до 127 по стандарту ASCII. Пример таблицы ASCII:

Dec	Hex	Char	Cmd
0	00		NUL
1	01	☺	SOH
2	02	☼	STX
3	03	♥	ETX
4	04	♦	EOT
5	05	♣	ENQ
6	06	♠	ACK
7	07	•	BEL
8	08	▣	BS
9	09	○	TAB
10	0A	▣	LF
11	0B	♂	VT
12	0C	♀	FF
13	0D	♪	CR
14	0E	🎵	SO
15	0F	☼	SI
16	10	▶	DLE
17	11	◀	DC1
18	12	↑	DC2
19	13	!!	DC3
20	14	¶	DC4
21	15	§	NAK
22	16	—	SYN
23	17	↕	ETB
24	18	↑	CAN
25	19	↓	EM
26	1A	→	SUB
27	1B	←	ESC
28	1C	└	FS
29	1D	↔	GS
30	1E	▲	RS
31	1F	▼	US

Dec	Hex	Char	Cmd
32	20		(sp)
33	21	!	
34	22	"	
35	23	#	
36	24	\$	
37	25	%	
38	26	&	
39	27	'	
40	28	(
41	29)	
42	2A	*	
43	2B	+	
44	2C	,	
45	2D	-	
46	2E	.	
47	2F	/	
48	30	0	
49	31	1	
50	32	2	
51	33	3	
52	34	4	
53	35	5	
54	36	6	
55	37	7	
56	38	8	
57	39	9	
58	3A	:	
59	3B	;	
60	3C	<	
61	3D	=	
62	3E	>	
63	3F	?	

Dec	Hex	Char	Cmd
64	40	@	
65	41	A	
66	42	B	
67	43	C	
68	44	D	
69	45	E	
70	46	F	
71	47	G	
72	48	H	
73	49	I	
74	4A	J	
75	4B	K	
76	4C	L	
77	4D	M	
78	4E	N	
79	4F	O	
80	50	P	
81	51	Q	
82	52	R	
83	53	S	
84	54	T	
85	55	U	
86	56	V	
87	57	W	
88	58	X	
89	59	Y	
90	5A	Z	
91	5B	[
92	5C	\	
93	5D]	
94	5E	^	
95	5F	_	

Dec	Hex	Char	Cmd
96	60	`	
97	61	a	
98	62	b	
99	63	c	
100	64	d	
101	65	e	
102	66	f	
103	67	g	
104	68	h	
105	69	i	
106	6A	j	
107	6B	k	
108	6C	l	
109	6D	m	
110	6E	n	
111	6F	o	
112	70	p	
113	71	q	
114	72	r	
115	73	s	
116	74	t	
117	75	u	
118	76	v	
119	77	w	
120	78	x	
121	79	y	
122	7A	z	
123	7B	{	
124	7C		
125	7D	}	
126	7E	~	
127	7F	␣	DEL

Обратите внимание на поля Dec и Char, каждому символу соответствует свой номер. Поэтому вы можете легко преобразовать какой-либо символ Char в его целочисленное значение.

```
julia> c = Int('x') # с помощью функции Int можно полу-  
чить номер элемента 'x'
```

```
-----
```

```
120
```

```
julia> typeof(c)
```

```
-----
```

```
Int64
```

Соответственно можно преобразовывать и в обратную сторону:

```
julia> Char(14)
```

```
-----
```

```
␣
```

Вы можете выполнять сравнения и некоторые арифметические операции с Char значениями (При чем операции будут происходить в соответствии с порядковыми номерами символов в таблице ASCII):

```
julia> 'A' < 'a'
```

```
-----
```

```
true
```

```
julia> 'A' <= 'a' <= 'Z'
```

```
-----
```

```
false
```

```
julia> 'A' <= 'X' <= 'Z'
```

```
-----
```

```
true
```

```
julia> 'x' - 'a'
```

```
-----
```

```
23
```

```
julia> 'A' + 1
```

'B'

Тип string

Значения типа String – набор (массив) элементов типа Char

```
julia> str = "Hello, world.\n"
```

```
-----
```

```
"Hello, world.\n"
```

```
julia> """Contains "quote" characters"""
```

```
-----
```

```
"Contains \"quote\" characters"
```

Длинные строки в строках можно разбить, поставив перед новой строкой обратную косую черту (\):

```
julia> "This is a long \
line"
```

```
-----
```

```
"This is a long line"
```

От строк возможно брать символы по диапазону (т.е. по ним возможна итерация):

```
julia> str[begin] # благодаря ключевому слову begin,
возможно получить первый элемент строки
```

```
-----
```

```
'H'
```

```
julia> str[1]
```

```
-----
```

```
'H'
```

```
julia> str[6]
```

```
-----
```

```
','
```

```
julia> str[end] # при помощи end можно получить послед-
ний
```

```
-----
```



```
'\n'
```

Вы также можете извлечь подстроку с помощью индексации диапазона:

```
julia> str[4:9]
```

```
-----
```

```
"lo, wo"
```

Обратите внимание, что выражения `str[k]` и `str[k:k]` не дают одинакового результата:

```
julia> str[6]
```

```
-----
```

```
','
```

```
julia> str[6:6]
```

```
-----
```

```
","
```

Первое представляет собой символьное значение типа `Char`, в то время как второе представляет собой строковое значение `String`, которое содержит только один символ. В Julia это разные вещи.

Julia полностью поддерживает символы и строки Юникода.

Задания для самостоятельного решения

- 1) Вывести на экран символы, соответствующие целочисленным значениям 15, 29, 74, 126.
- 2) Присвоить строковой переменной значение, совпадающее с Вашим ФИО, и вывести на экран 4 и 11 символ получившейся строки.
- 3) Выделить из строки «Без труда не выловишь и рыбку из пруда» подстроку [5:9].
- 4) В строке «Ткрпкник и» заменить все буквы 'к' на 'е' и вывести на экран.

- 5) Сложить (объединить) строки, полученные в 3 и 4 задании. Полученную строку сложить со строкой « – всё перетрут».
- 6) Вывести на экран число символов и число байт в строке, полученной в 5 задании.

Условные выражения в языке программирования Julia

Условные конструкции выглядят следующим образом:

```
if условие
    Выражение
end
```

Наиболее распространенным условным выражением в Julia является if ... else.

```
if условие
    Выражение
else
    выражение
end
```

Рассмотрим простой пример:

```
print("Введите a: ")
a = parse{Int, readline()}
if a > 0
    print(a)
else
    print(a + 1)
end
```

```
Введите a: 2
-----
```

2

Здесь мы видим, что условие `if a > 0` является истинным, следовательно выполняется первая ветвь и в результате выполнения возвращается значение `print(a)`, которое равно 2.

```
Введите a: -1
```

0

Здесь условие `if a > 0` ложно, поэтому берется вторая ветвь. Возвращаемое значение `print(a + 1)`, которое равно 0.

Также есть конструкция с оператором `elseif`:

```
if условие1
    выражение1
elseif условие2
    выражение2
else
    выражение3
end
```

`elseif` в условной конструкции может быть сколько угодно. Посмотрим пример:

```
print("Введите a: ")
a = parse(Int, readline())
if a > 0
    print(a)
elseif a == 0
    print(a + 1)
else
    print(a - 1)
end
```

Введите a: -1

-2

Здесь выполняется третья ветвь, возвращается значение `print(a - 1)`, равное -2

Введите a: 0

1

Здесь выполняется вторая ветвь, возвращается значение `print(a + 1)`, равное единице

```
Введите a: 1
-----
1
```

Тут выполняется первая ветвь, возвращается значение `print(a)`, равное 1.

Тернарный (или троичный) оператор

Синтаксис:

```
условие ? выражение1 (выполняется если условие истинно)
: выражение2 (выполняется если условие ложно)
```

```
a ? b : c
```

Тернарный оператор `?` тесно связан с синтаксисом *if else*, но в основном используется в случаях, когда необходимо присвоить переменной значение согласно какому либо условию.

Выражение *a* перед `?` выражением является выражением условия, и троичная операция вычисляет выражение *b* перед `:`, если условие *a* равно *true*, или выражение *c* после `:`, если оно равно *false*. Обратите внимание, что пробелы вокруг `?` и `:` являются обязательными: выражение вида *a?b:c* не является допустимым троичным выражением (но перенос строки допустим как после `?` так и после `:`).

Рассмотрим несколько примеров использования тернарного оператора.

```
x = 1; y = 2;
yes = "x меньше y"
no = "x больше y"
x < y ? yes : no
-----
"x меньше y"
```

Итак, программа работает, однако не совсем корректно, так как нет вывода о том, что переменные равны.

Модифицируем прошлую программу, добавив недостающий вывод:

```
x = 2; y = 2;
less = "x меньше y"
more = "x больше y"
```

```
equal = "x равно y"  
x > y ? more : x < y ? less : equal  
-----  
"x равно y"
```

Теперь программа работает полностью корректно.

Задачи для самостоятельного решения.

1. Дано два числа. Вывести на экран наибольшее из чисел;
2. Пользователь вводит два числа с клавиатуры. Вывести на экран yes, если они отличаются друг от друга на 135, иначе вывести на экран No (Использовать тернарный оператор);
3. Дано число. Если оно больше 100 или меньше -100, то вывести на экран символ "—", иначе вывести на экран символ "+" (Использовать тернарный оператор);
4. Пользователь вводит номер месяца (от 1 до 12). Вывести название сезона года на экран (зима, весна, лето, осень);
5. Пользователь вводит три числа. Если все числа больше 10, то вывести на экран yes, иначе no (Использовать тернарный оператор);
6. Пользователь вводит три целых числа. Два из них равны друг другу. Выведите третье число, не равное остальным. Если среди введенных чисел не оказалось двух равных друг другу, выведите строку "Ошибка";
7. Пользователь вводит целое число. Проверьте является ли это число четырехзначным, если является, то выведите строку "Успешно", иначе "Неудача";
8. Пользователь вводит целое число - текущее время в часах. Если количество часов находится между 7 и 10, то программа должна вывести строку "Пора вставать!", иначе выведется строка "Ты проспал!". Если введенное число отрицательно или больше 23, то программа должна вывести строку "Ошибка".

Составные выражения на языке программирования Julia

Иногда удобно иметь одно выражение, которое вычисляет несколько подвыражений по порядку, возвращая значение последнего подвыражения в качестве его значения. Для этого существуют две конструкции Julia: *begin* блоки и *;* цепочки. Значение обеих конструкций составного выражения равно значению последнего подвыражения. Вот пример *begin* блока:

```
z = begin
    println("Введите x: ")
    x = parse{Int, readline()}
    println("Введите y: ")
    y = parse{Int, readline()}
    println("x + y = ")
    x + y
end
```

```
Введите x:
3
Введите y:
4
x + y =
-----
7
```

Поскольку это довольно небольшие и простые выражения, их можно легко поместить в одну строку, и именно здесь «*;*» пригодится синтаксис цепочки:

```
z = (x = 1; y = 2; x + y)

-----
3
```

Хотя это типично, нет требования, чтобы *begin* блоки были многострочными или чтобы *;* цепочки были однострочными:

```
begin z = x = 1; y = 2; x + y end
```

3

Функции в Julia

В Julia функция – это объект. Функции в Julia могут возвращать любой тип данных. Называть функции можно по таким же правилам как у переменных. Основной синтаксис для определения функций в Julia:

```
function f(x, y)
    return x^y
end
```

Данная функция принимает два аргумента x и y и возвращает значение x возведенного в степень y , стоит отметить, что как только программа дойдет до ключевого слова `return` выполнение функции закончится.

Есть еще альтернативный способ задания функции:

```
f(x, y) = x^y
```

Синтаксис вызова функции выглядит следующим образом:

```
f(x, y)
```

Реализуем следующую программу на Julia. Будет дано два целых числа a , b необходимо вывести на экран большее число. (Мы реализуем это при помощи функции, чуть позже будет понятно, зачем они нам нужны):

```
function f(a, b)
    if a > b
        return a
    else
        return b
    end
a = 5;
b = -7;
print(f(a, b))
```

```
-----
5
```

Так как функции это объекты, то их можно присваивать друг у другу:

```
g = f;
print(g(3, 9))
```

```
-----
```


Возвращаемый тип может быть задан явно — это преобразует указанный возвращаемое значение в указанный тип, делается это при помощи ключевого оператора: `::`

Пример:

```
function multy(x, y)::Float32
    return x * y
end;
```

```
print(typeof(multy(2, 3)))
```

```
-----
Float32
```

Итак, а теперь давайте напишем следующую программу: Нам будет известна некая функция $f(x)$ и $g(x)$, необходимо посчитать $f(x) + g(x)$, $f(x) - g(x)$, $f(x) * g(x)$, $f(x) / g(x)$:

```
f_x(x) = x^2 - 2*x + 4
```

```
g_x(x) = -x^2 - 2*x + 4
```

```
x = 3
```

```
print(f_x(x) + g_x(x))
```

```
print(f_x(x) - g_x(x))
```

```
print(f_x(x) * g_x(x))
```

```
print(f_x(x) / g_x(x))
```

```
-----
```

```
-4
```

```
18
```

```
-77
```

```
-0.63636363
```

Функции нужны, чтобы заметно упрощать и сокращать код, адаптировать его для разных платформ, делать более отказоустойчивым, легко отлаживать. И вообще порядок в функциях — порядок в голове.

Область действия переменных в функции

Область действия переменной - это область кода, в пределах которой доступна переменная. Разграничение области действия переменных помогает избежать конфликтов именования переменных. Подробнее об областях видимости мы поговорим позже, а пока условимся, что переменные по области видимости делятся на **локальные (local)** и **глобальные (global)**.

Локальные переменные – те переменные, которые видны только внутри определенного блока кода, например, функции.

Глобальные переменные – переменные, видимые из любого блока кода.

Рассмотрим пример:

```
function greet()  
    x = "hello" # локальная переменная  
    println(x)  
end  
-----  
  
greet (generic function with 1 method)  
  
julia> greet()  
-----  
hello  
  
julia> x # попытка вызвать x вне функции  
-----  
ERROR: UndefVarError: x not defined
```

При попытке вызвать переменную `x` компилятор выдает ошибку, потому что переменная `x` существует только в области видимости самой функции, это значит, что мы можем вызывать или как-либо использовать ее только внутри функции.

Теперь рассмотрим ситуацию, когда в глобальной области видимости уже объявлена переменная `x`:

```
julia> x = 123 # global  
123  
  
julia> function greet()  
    x = "hello" # new local  
    println(x)  
end
```

```
greet (generic function with 1 method)
```

```
julia> greet()  
hello
```

```
julia> x # global  
123
```

Поскольку x в функции `greet()` является локальным его вызов никак не влияет на значение глобального x .

Рассмотрим следующий пример:

```
julia> function sum_to(n)  
    s = 0 # new local  
    for i = 1:n  
        s = s + i # assign existing local  
    end  
    return s # same local  
end
```

```
-----  
sum_to (generic function with 1 method)
```

```
julia> sum_to(10)  
-----  
55
```

```
julia> s # global  
-----  
ERROR: UndefVarError: s not defined
```

Как и в предыдущем примере, первое присвоение s в начале `sum_to` приводит к созданию новой локальной переменной в теле функции. Цикл `for` имеет свою собственную внутреннюю локальную область видимости в пределах области видимости функции. s уже есть локальная переменная, поэтому присвоение обновляет существующую s вместо создания новой локальной.

Поскольку s является локальной для функции `sum_to`, вызов функции не оказывает никакого влияния на глобальную переменную s . Поэтому вызывать s в глобальной области видимости нельзя.

Задачи для самостоятельного решения.

1. Реализовать функцию, которая принимает 2 целочисленных аргумента x и y и возвращает следующее выражение: $x^4 - y^3 + x/y$.
2. Реализовать функцию, которая принимает 1 значение типа `string` и возвращает строку, которая будет состоять из символов стоящих на четных местах аргумента.
3. Реализовать 2 функции $f(x) = x^2 + x - 3$ и $g(x) = x^3 - x + 1$. Посчитать сумму произведений функций $f(x)$ и $g(x)$ при x из промежутка целых чисел от 1 до 120.
4. На вход подается целое, необходимо реализовать 3 функции: первая - должна вернуть длину числа, вторая – произведение цифр числа, то есть, если введено число 56, то программа должна вернуть произведение 5 и 6 – это 30, третья – сумму цифр числа.

Массивы

Массив - это структура данных хранящая набор значений (элементов массива), идентифицируемых по индексу или набору индексов. Различают одномерные массивы, которые можно рассматривать как реализацию абстрактного типа данных вектор, двумерные: имеют представление матрицы и многомерные массивы.

В последующем в рамках данного курса говоря о массивах будем иметь ввиду одномерные

Способы объявления массивов

В языке Julia существует довольно много способов объявления массива. Рассмотрим некоторые из них.

Простейший способ неявного объявления пустого одномерного массива: `a[]`, за счёт динамической типизации язык сам определяет тип данных при выполнении сценария(код программы).

Явное объявление массивов:

- `a=String[]`
- `b=Int32[]`
- `c=Float32[]`

Где `a`, `b`, `c` - пустые массивы соответствующих типов.

Также массивы можно задавать по средствам вызова встроенных функций:

`Array{T}(undef, dims)` - Массив типа `T` и размерности `dims`.

Пример: присвоим переменной `a` массив целочисленного типа данных(`Int`) с 3-мя элементами

```
a=Array{Int}(undef, 3)
```

При такой инициализации массива элементы в нём не определены т. е. в нём хранится (мусор), о чём не стоит забывать при данном способе задания.

`zeros(T, dims)` - Массив нулей

`ones(T, dims)` - Массив единиц

В Julia индексация массивов начинается с единицы

Доступ к элементам можно осуществить двумя способами:

1) $a[1], a[2], a[3], \dots$

2) $a[1, 1], a[1, 2], a[1, 3], \dots$

Второй способ справедлив, так как при объявлении одномерного массива в Julia создаётся матрицу с одним строкой и указанным количеством столбцов.

Определённые операции над массивами

Работа с элементами массива:

Добавить элемент v в конец массива a можно с использованием функции: ***push!(a,v)***

Наличие символа "!" в конце имени функции обычно служит указанием на то, что первая переменная в списке параметров будет изменена.

Пример:

```
a=[1, 2, 3, 4]
push!(a, 5)
```

```
a=[1, 2, 3, 4, 5]
```

Добавить элемент v в начало массива a можно с использованием функции: ***pushfirst!(a,v)***

Пример:

```
a=[1, 2, 3, 4]
pushfirst!(a, 5)
```

```
a=[5, 1, 2, 3, 4]`
```

Для массива *a* типа *Vector* определена операция вставки элемента *x* на произвольную позицию *n*: ***insert!(a,n,x)***

Пример:

```
print(insert!([1, 2, 3, 4], 2, 8))  
-----  
[1, 8, 2, 3, 4]
```

Последний элемент массива *a* можно удалить так: ***pop!(a)***

Пример:

```
a=[1, 2, 3, 4]  
pop!(a)  
print(a)  
  
-----  
[1, 2, 3]
```

Первый элемент массива *a* можно удалить так: ***popfirst!(a)***

Пример:

```
a=[1, 2, 3, 4]  
popfirst!(a)  
print(a)  
  
-----  
[2, 3, 4]
```

Удалить элемент массива в позиции *pos* можно так: ***deleteat!(a, pos)***

Пример:

```
a=[1, 2, 3, 4]  
deleteat!(a, 3)  
print(a)  
  
-----  
[1, 2, 4]
```

Поменять порядок элементов массива на обратный: ***a=a[end:-1:1]***

Пример:

```
a=[1, 2, 3, 4]
a=a[end: -1: 1]
print(a)
-----
[4, 3, 2, 1]
```

Проверить, есть данный элемент x в массиве: ***in(x, a)***

Пример:

```
a=[1, 2, 3, 4]
in(5, a)
print(a)
-----
false
```

Определить длину массива: ***length(a)***

Пример:

```
a=[1, 2, 3, 4]
length(a)
-----
4
```

Определить размер массива в байтах: ***sizeof(a)***

Пример:

```
a=[1, 2, 3, 4]
sizeof(a)
-----
32
```


Найти максимальное число массива: *maximum(a)*

Пример:

```
a=[1, 2, 3, 4]
maximum(a)
```

```
-----
4
```

Найти минимальное число массива: *minimum(a)*

Пример:

```
a=[1, 2, 3, 4]
minimum(a)
```

```
-----
1
```

Выборки (slice)

Выбрать элементы массива(вектора) *a* с *begin* по *end* с шагом *step*:

```
a[begin: end: step]
```

Если явно не указывать шаг, то он равен единицы.

Пример:

Выбрать элементы вектора *a* с 3 по 5:

```
a=[1, 2, 3, 4, 5]
b=a[3: 5]
print(b)
```

```
-----
[3, 4 ,5]
```

Пример:

Можно сделать выборку с шагом: `a[1:3:10]`

```
a=[1, 2, 3, 4, 5]
b=a[1: 3: 2]
print(b)
```

```
-----
[1]
```

Задачи для самостоятельного решения.

1. Дан одномерный целочисленный(Int) массив, состоящий из n-го количества элементов(ввод кол-ва элементов осуществляется с клавиатуры). Нужно вывести на экран все простые числа, содержащиеся в массиве.

Примечание: Если таких чисел в массиве не содержится вывести предупреждение об этом.

2. Дан одномерный целочисленный(Int) массив, состоящий из n-го количества элементов(ввод кол-ва элементов осуществляется с клавиатуры). Нужно проверить все элементы на условие: является ли число совершенным, и вывести на экран индексы таковых последовательно.

Примечание: Если таких чисел в массиве не содержится вывести предупреждение об этом.

3. Дан одномерный вещественный(Float) массив, состоящий из n-го количества элементов(ввод кол-ва элементов осуществляется с клавиатуры) в котором нужно найти два минимума(min). То есть после нахождения минимума необходимо найти ещё один среди остальных элементов и вывести результаты на экран.

Примечание: Нельзя использовать встроенные функции, такие как поиск min/max и сортировки. Так же запрещено использовать собственные функции сортировок и вспомогательные массивы.

4. Дан одномерный целочисленный(Int) массив, состоящий из n-го количества элементов (ввод кол-ва элементов осуществляется с клавиатуры) в котором нужно осуществить чередование нечётных с чётными элементами, а именно: первым идёт нечётный, после - чётный и т. д. После вывести на экран массив с данной последовательностью.

Примечание: Нельзя использовать вспомогательные массивы.

5. Дан одномерный вещественный(Float) массив, состоящий из n-го количества элементов(ввод кол-ва элементов осуществляется с клавиатуры), который нужно отсортировать таким образом, чтобы с первого по центральный элемент элементы шли в порядке возрастания, а с центрального по последний элемент - в порядке убывания.

Примечание: Нельзя использовать встроенные функции сортировок и без использования вспомогательных массивов.

Кортежи

Кортеж — группа значений некоторых величин фиксированной длины, разделенных запятыми. Кортеж в языке программирования Julia задается с помощью круглых скобок, например:

```
c=(1, 1.1, pi, 'c', "Julia", 1//3)
```

Кортеж — это контейнер данных который может содержать элементы разных типов данных, в отличие от массива, который является контейнером однородных данных.

- Кортеж можно превратить в вектор типа Any: `v=[c...]`
- Вектор можно превратить в кортеж: `c=(v...,)`
- Элементы кортежа неизменяемы, оператор `c[1]=2` вызовет сообщение об ошибке.
- Можно использовать именованный кортеж вида `c=(a=1, b=2.2)`
Доступ к элементам кортежа либо через индекс, например, `c[1]`, либо по имени `c.a`, `c.b`.

Если список возвращаемых величин какой-либо функции содержит несколько разнородных значений, то возвращаются они в форме кортежа.

Словари

Словарь — набор значений состоящих из пар элементов ключ-значение. Словарь создается с помощью функции `Dict()`:

`d = Dict()` — пустой словарь, а словарь со значениями можно задать следующим образом:

```
d = Dict{'a'=>1, 'b'=>2, 'c'=>3}
```

Заполнить словарь символами алфавита и их номерами:

```
y=collect('a':'z')
d = Dict{<
for i in 1:length(y)
    d[y[i]]=i
end
```

- Информацию в словарях можно менять, например:

```
d['a']=100; d['a']+=1
```

- Можно добавить пару ключ-значение:

```
d[key] = value
```

- Так же можно найти все значения по ключу:

```
d['a']
```

- Получить список всех ключей поможет функция:

```
keys(d)
```

- Получить список значений функция:

```
values(d)
```

- Проверить наличие ключа в словаре возможно с помощью функции:

```
haskey(d, 'a')
```

Пример:

Распечатать словарь (ключ-значение)

```
for (k,v) in d
    println("$k - $v")
end
```

Последовательность элементов словаря не фиксируется, т. е. может быть произвольной.

Задачи для самостоятельного решения

1. Написать программу на языке julia, выполняющую следующую операцию. Нужно написать функцию которая сортирует все элементы внутри словаря по их количеству в словаре.

Циклы `for`, `while`

В программировании существуют задачи, которые подразумевают повторное выполнение действий. Для того, чтобы не загромождать код, используют специальные конструкции, которые выполняют действие либо определенное количество раз, либо до определенного условия, которое проверяется на каждом "шаге". Существует две конструкции для повторного вычисления выражений: цикл **`while`** и цикл **`for`**.

Цикл `for`

Синтаксис оператора **`for`** аналогичен определению функции. У него есть заголовок и тело, которое заканчивается ключевым словом **`end`**. Тело может содержать любое количество утверждений.

Рекомендуется все действия и операции в теле цикла писать с двойным отступом, чтобы показать, что они находятся внутри цикла **`for`**. Так, благодаря циклу **`for`** можно переписать код ниже более кратко:

```
# вариант без цикла for
```

```
println(1)
println(2)
println(3)
println(4)
println(5)
-----
1
2
3
4
5
```

```
# вариант с циклом for
```

```
for i = 1:5
    println(i)
end
-----
1
2
3
4
```

Здесь `1:5` - это диапазон, представляющий последовательность чисел 1, 2, 3, 4, 5. Цикл **for** перебирает эти значения, присваивая каждое из них по очереди переменной `i`.

В общем случае конструкция цикла `for` может выполнять итерацию над любым объектом. В таких случаях вместо знака равно (`=`) обычно используется альтернативное (но полностью эквивалентное) ключевое слово **in**, поскольку оно делает код более понятным:

```
for number in [1,2,3,4,5]
    println(number)
end
-----
1
2
3
4
5
```

Не всегда надо перебирать каждый элемент, в связи с чем появляется необходимость пробегать по последовательности чисел с шагом, отличным от 1. Так, в случае, если необходимо вывести каждое 3-е число из диапазона `1:10`, достаточно при объявлении цикла **for** указать дополнительный параметр, отвечающий за шаг. В цикле **for** шаг указывают между начальным значением и конечным. Например:

```
for j=1:3:10
    println(number)
end
-----
3
6
9
```

Область видимости переменных внутри циклов

Циклы `for` и `while` вводят новые области видимости. Как уже оговаривалось ранее, область видимости переменной – это область кода, в которой переменная доступна.

Области видимости делятся на:

- Локальную
- Глобальную

При работе с локальной областью видимости все переменные, которые были объявлены до цикла *for* и не входящие во вложенные конструкции будут иметь глобальную область видимости относительно тела цикла *for* и к ним можно будет обратиться внутри него, но не все так просто.

Например:

```
for j = 1:5
    println(j) # локальное обращение
end
println(j) # глобальное, выдаст ошибку
-----
1
2
3
4
5
ERROR: UndefVarError: j not defined
```

Для того, чтобы изменить внутри цикла *for* значение переменной, которая была объявлена до самого цикла, необходимо использовать специальную команду ***global***:

```
counter = 0
for number in [1,2,3,4,5,6]
    if number % 2 != 0
        global counter += 1
    end
end
println(counter)
-----
3
```

Оператор ***global*** сообщает интерпретатору, что в цикле, при использовании *counter*, речь идет о глобальной переменной и локальную создавать не надо. Однако оператор ***global*** не нужен, если оператор ссылается на **неизменяемый** тип данных.

Цикл while

Помимо цикла for так же активно используют еще один цикл – **while**.

Пример цикла **while**:

```
i = 1
while i <= 5 #условие
    println(i) #тело цикла
    global i+=1
end
-----
1
2
3
4
5
```

Цикл while будет выполняться до тех пор, пока условие остается истинным.

Иногда бывает удобно прервать повторение цикла while до того, как будет нарушено условие проверки, или остановить итерацию и выйти из цикла for до того, как будет достигнут конец итерируемого объекта. Этого можно добиться с помощью ключевого слова **break**:

```
# пример 1
i = 1
while true
    println(i)
    if i >=5
        break
    end
    global i+=1
end
-----
1
2
3
4
5
```

```
# пример 2
for j = 1:1000
    println(j)
    if j >= 5
        break
    end
end
```

```

    end
end
-----
1
2
3
4
5

```

Без ключевого слова **break** приведенный выше цикл **while** никогда бы не завершился сам по себе, а цикл **for** выполнял бы итерации до 1000. Эти циклы завершаются досрочно с помощью **break**. В других случаях удобно иметь возможность не доводить итерацию до конца и сразу перейти к следующей. Для этого используется ключевое слово **continue**:

```

for I = 1:10
    if I % 3 != 0
        continue
    end
    println(i)
end
-----
3
6
9

```

Задачи для самостоятельного решения

- Для заданного массива [13, 2, 23, 4, 67, 19, 45, 5, 16, 1, 11, 127, 14] выполните следующие действия используя оба цикла
 - Выведите массив на экран
 - Выведите каждый второй элемент
 - Выведите массив в обратном порядке
 - Выведите в обратном порядке каждый 3 элемент массива
- Используя массив из предыдущего номера, сложите каждый четный элемент массива и вычтите каждый второй нечетный элемент.

3. Для массивов `[0,1,2,3,4,5,6,7,8,9]` и `[0,1,2,3,4,5,6,7,8,9,10]` выведите на экран только центральный элемент массива. В случае, если элементов в массиве чётное число, выведите 2 таких числа.
4. Для массивов `numbers = [0,1,2,3,4,5,6,7,8,9]` и `degrees = [0,1,2,3,4,5,6,7,8,9]` напишите программу так, чтобы получить следующий результат: `[0,1,128,729,1024,625,216,49,8,1]`.

Векторизованные операции

Векторизованные операции работают с массивами поэлементно. Что-бы векторизовать операцию, нужно написать перед ней точку (`.+` ; `.-` ; `.^` ; `.*`)

Пример:

```
a = [1, 2, 3]
a .+= 1
print(a)
-----
[2, 3, 4]
```

Векторизация помогает избавиться от циклов в некоторых задачах, такой подход занимает меньше строчек в коде и ускоряет работу программы.

Векторизацию можно использовать для поэлементных операций между массивами.

Пример:

```
a = [1, 2, 3]
b = [2, 5, 3]
print(a .* b)
-----
[2, 10, 9]
```

Также можно векторизовать встроенные функции и функции написанные пользователем, для этого нужно после имени функции добавить точку (`sin.(a)` и `log.(a)`):

```
a = [1, 2, 3]
print(sin.(a))
```

```
-----  
[0.8414709848078965, 0.9092974268256817, 0.1411200080598672]
```

Так же векторизацию можно применять для самописных функций:

```
function m(elem)  
    if elem < 0  
        elem = 0  
    end  
elem  
end
```

```
a = [1,-2,3,-4]  
print(m.(a))
```

```
-----  
[1,0,3,0]
```

Задачи для самостоятельного решения

1. Написать программу на языке julia, выполняющую следующую операцию. Нужно задать 2 массива A и B одинаковой размерности n (задаётся пользователем), провести над ними операцию $2 \cdot A^3 + 3 \cdot B^2$. Написать нужно программу сначала не используя векторизацию, а после используя её.
2. Написать программу на языке julia, выполняющую следующую операцию. Нужно задать массив размерности n (задаётся пользователем), и используя векторизацию изменить все элементы массива которые равны первому элементу на ноль. Эта операция над массивом должна быть написана в одну строчку кода.

Работа с файлами в Julia.

Типы файлов

Существуют два основных типа файлов: текстовые и двоичные.

Текстовые файлы - это файлы, состоящие из любых символов.

Они состоят из строк, каждая из которых в конце имеет символ "конец строки".

А сам файл завершается символом "конец файла".

Двоичные файлы - это файлы, состоящие из блоков определенного размера.

В блоках могут храниться данные любого вида и структуры.

Если в файле хранятся только числа или данные определённой структуры, то для хранения таких значений удобно использовать именно двоичные файлы.

Работа с файлами

Работа с файлом начинается с того, что указывается его имя

```
filename = "/home/user/example.txt"
```

По умолчанию рабочим является каталог, в который установлена Julia. Поэтому следует указывать полные пути к файлам.

Далее файл нужно открыть *file* = *open(filename)*

Режим	описание
r	Чтение
w	Запись, создание, изменение размера
a	Запись, создание, дополнение
r+	Чтение, запись

По умолчанию функция *open(filename, mode="r")* открывает файлы только для чтения. Для других возможностей используются другие режимы.

w+	Чтение, запись, создание, изменение размера
a+	Чтение, запись, создание, дополнение

Очевидно, можно сразу выполнить команду
`file = open("/home/user/example.txt", "r+")`.

Проверить наличие нашего файла можно с использованием функции `ispath("/home/user/example.txt")`, которая возвращает `true`, если файл существует, или `false`, если такого файла нет.

Функция `readdir("/home/user")` выводит на экран все файлы, которые есть в данном каталоге. Например, вывести список файлов в текущем каталоге можно так `readdir(pwd())`, где `pwd()` вернет нам полный путь к текущему каталогу.

Можно прочитать все данные из файла сразу командой `data = readlines(file)`. В этом случае `data` содержит массив строк.

Для обработки информации файла построчно можно использовать цикл:

```
for line in data
  println(line)
  необходимые действия
end
```

Рекомендуется всегда закрывать файл после завершения работ с ним при помощи функции `close(file)`.

Наиболее простой способ работы с данными из файла с именем `filename` выглядит так:

```
open(filename) do file
  ... действия с файлом file
end
```

В этом случае файл закрывать не нужно, он закрывается автоматически после выхода из блока.

Если предполагается запись информации в файл, то его нужно открыть с ключом `"w"` и произвести запись при помощи функции `write()` или `println()`:

```
filename = "/home/user/example.txt"
open(filename, "w") do file
```

```
write(file, "Текст записан в файл\n")
println(file, "в том числе и командой println!")
end
```

Однако, если с ключом `"w"` открывается существующий файл, он будет перезаписан.

Ключ `"a"` позволяет открыть существующий файл с возможностью добавления информации.

А теперь посмотрим, как же работать с двоичными файлами, для этого запишем массив вещественных чисел `x` и прочитаем его из файла в массив `y` и сравним их.

Для чтения данных из двоичного файла используется функция `read!()`

Делается это следующим образом:

```
filename = "/home/user/example.bin"

x = rand(Float32, 10, 10, 10)
open(filename, "w") do file
    write(file, x)
end
y = Array{Float32}(undef, (10, 10, 10))
open(filename) do file
    read!(file, y)
end

x == y
-----
true
```

Решения задач