

# Учебная практика BASH для чайников

---

## Оглавление

- Учебная практика BASH для чайников
  - Оглавление
  - От автора
  - Глава I. Да кто этот ваш `.bash`?
  - Глава II. Строки — это всё
    - 2.1. Объявление переменных
    - 2.2. Чтение переменных или почему программистам так много платят
    - 2.3. Что там, под капотом?
    - 2.4. Несколько способов взаимодействия со строками
      - 2.4.1. Длина строки
      - 2.4.2. Приведение к определённому регистру
    - 2.5. Ввод переменных пользователем
    - 2.6. Кавычки? Не, не слышали
    - 2.7. Загадочный доллар
    - 2.8. Позиционные переменные
    - 2.9. Вопросы существования с точки зрения программиста
  - Глава III. Ветвления в bash-е или "А что, если..."
    - 3.1. Синтаксис
    - 3.2. Чем больше - тем лучше
    - 3.2. Социальное дистанционирование
    - 3.3. Вопросы бытия, часть II
    - 3.4. Advanced-level ветвлений
      - Логические операторы и эквивалентность:
      - `==` и `!=`
    - 3.5. Жизни пробелов тоже важны!
    - 3.6. Таинственные минусы
    - 3.7. `=~`
    - 3.8. Короткая запись условия
  - Глава IV. Что такое регулярные выражения и с чем их есть?
    - 4.1. Это база!
    - 4.2. Снова `[ ]`
    - 4.2. Кванторы
    - 4.3. Экранирование
    - 4.4. Повторения
    - 4.5. Группы
    - 4.6. Якоря
    - 4.7. Числа!
    - 4.8. С чем регулярные выражения путать нельзя или маски файлов
  - Глава V. Магия математики
    - 5.1. Школьная арифметика
    - 5.2. Возврат результата вычисления

- 5.3. Конкатенация - не сложение!
- 5.4. Математические условия
- Глава VI. Вот они, слева направо: `while`, `until`, `for`
  - 6.1. `while` и `until`
  - 6.2. Всемогуший `for`
    - 6.2.1. Для массивов
    - 6.2.2. Для простых последовательностей
    - 6.2.3. Си-стайл
  - 6.3. Обработка позиционных параметров

## От автора

Методическое пособие создано для потока 1-го курса ИУ7 2023 года для помощи в освоении предмета ПТП. Автор допускает, что в методическом пособии присутствуют ошибки, поэтому о них следует сообщать по контактам ниже:

- telegram: [@zhikhkirill](#)
- VK: [@zhikh.localhost](#)

## Глава I. Да кто этот ваш `.bash`?

Программисты — люди ленивые, привыкшие все автоматизировать. Поэтому неудивительно, что появилась такая вещь, как bash-скрипты (далее автор будет называть bash-скрипты просто **bash**). В bash можно записывать целые сценарии для выполнения рутинной работы. Приведу пример:

```
#!/bin/bash
# test.sh

for file in "./tests/in/*"; do
    cat $file | python3 my_lab.py > ${file/in/out}
done
```

Скрипт автоматически вводит данные в Вашу программу (например, лабораторную по программированию) и записывает результаты в файл. Таким образом Вы можете протестировать свою программу сразу на нескольких массивах входных данных *одной командой!*

```
bash test.sh
```

Неплохо, не правда ли? А ведь это только начало! Возможности BASH-скриптов ограничены лишь Вашим воображением.

И да, `bash` есть на `Linux`-системах. Предмет ПТП подразумевает, что у Вас уже есть данная ОС в любом виде.

Перейдём к основному материалу.

## Глава II. Строки — это всё

В bash-е нет типов переменных!

Просто запомните это. Каждый раз, когда Вы используете переменную, помните эти слова. Чтобы лучше это понять, я задам Вам вопрос: а есть ли типы данных в языке программирования Python (далее — просто Python)? Ведь мы можем сделать так:

```
a = 5
a = "foo"
```

Как будто у переменной `a` нет определённого типа. Однако это не означает, что в Python нет типов данных. Просто *переменная не имеет строго определённого типа*, т.к. в этом ЯП (язык программирования) *динамическая типизация*.

```
>>> a = "foo"
>>> type(a)
<class 'str'>
>>> a = 5
>>> type(a)
<class 'int'>
```

А вот с bash-ем все не так:

В bash все переменные являются строками!

Исходя из этого утверждения будет строиться вся логика работы bash-a.

## 2.1. Объявление переменных

Все просто, например переменная `var` со значением 5:

```
var=5
```

Обязательно без пробелов!

```
var=5      # CORRECT
var = 5    # INCORRECT
var= foo   # INCORRECT
var =bar   # INCORRECT
```

## 2.2. Чтение переменных или почему программистам так много платят

Просто потому, что они используют символ доллара `$` для чтения переменных:

```
a=5
echo $a      # 5
```

Иногда нужно чётко ограничить название переменной, например:

```
a=foo
abar="i wanted something else..."
echo $abar      # i wanted something else...
echo ${a}bar    # foobar
```

Как вывести знак доллара, если он означает чтение переменной? Используем экранирование:

```
a=5
echo $a
echo \$a
```

Вывод:

```
5
$a
```

## 2.3. Что там, под капотом?

Чтобы понять, как работают переменные под капотом, рассмотрим такой код:

```
a="hello"
b="echo $a"
$b
```

Интерпретатор в большинстве случаев не работает с переменными напрямую. Он проходит по строке и заменяет все указания переменных на их соответствующее значение, а *потом* исполняет строку как код.

```
    b="echo $a"    # source code
#      |
#  $a  -> hello
#      |
#      v
    b="echo hello" # command
```

```
        $b        # source
#      |
#  $b  -> echo hello
#      |
#      v
    echo hello    # command
```

Именно поэтому Вы могли часто замечать ошибку `bash: <sth>: not found`. Интерпретатор подставляет на место переменной её значение и пытается выполнить как команду.

## 2.4. Несколько способов взаимодействия со строками

### 2.4.1. Длина строки

```
a=abc
echo ${#a}    # 3
```

### 2.4.2. Приведение к определённому регистру

```
a=Abc
echo ${a,,}    # abc
echo ${a^^}    # ABC
```

## 2.5. Ввод переменных пользователем

Если переменные вводятся построчно:

```
read a
read b
```

Если через пробел:

```
read a b
```

Обратите внимание: никаких `$` не нужно!

Если нужно ввести *не* через пробел, а, например, через запятую и пробел, можно изменить разделитель:

```
IFS=", "
```

Хорошей практикой будет сохранять значение предыдущего разделителя:

```
OLD_IFS=$IFS
IFS=", "
```

```
read a b

IFS=$OLD_IFS
```

## 2.6. Кавычки? Не, не слышали

Автор часто замечал такую запись:

```
# WRONG
a="foo"
```

Или:

```
# WRONG
echo "${a}"
```

Однако кавычки в этих случаях *избыточны*. Нам ничто не мешает сделать так:

```
# CORRECT
a=foo
echo $a
```

Потому что вне зависимости от наличия кавычек, переменная всё равно будет, и будет она *строковой*.

Но если Вам нужно создать переменную с пробелом внутри, то кавычки обязательны!

```
a="foo bar"
```

Аналогично для вывода данных через `echo`. Кавычки в большинстве случаев можно опускать:

```
a=foo
b=bar
echo "$a $b"
echo $a $b
```

Вывод будет:

```
foo bar  
foo bar
```

Вот случай, когда без кавычек никак:

```
echo "$a "
```

А как вывести строку с кавычками? Применяем экранирование:

```
a=foo  
b=bar  
echo "\"$a $b\""
```

Тогда вывод будет:

```
"foo bar"
```

Не нужен перевод на новую строку? Делаем так:

```
echo -n "some"  
echo " text"
```

Вывод:

```
some text
```

## 2.7. Загадочный доллар

В bash-е есть уникальные переменные. Одна из них - это `$?`. Её значение равно *коду возврата последней выполненной команды*. Например:

```
some_program  
echo $?
```

Код возврата равен 0, если программа завершилась успешно.

Поэтому команда `true` (да, есть такая команда) всегда возвращает 0, а `false` - 1.



```
true
echo $?      # 0
false
echo $?      # 1
```

Зачем это надо? Вернёмся к `$?` в будущем.

## 2.8. Позиционные переменные

Ещё один тип особых переменных. Увидим их в действии. Запустим такой код:

```
# test.sh
echo \${0}=${0}
echo \${1}=${1}
```

Запускаем вот так:

```
your-directory@username$ bash test.sh hello
```

Получаем ответ:

```
${0}=your-directory/test.sh
${1}=hello
```

Как Вы заметили, `$n`, где `n` — неотрицательное число, это ничто иное, как аргументы при запуске скрипта. Причём первый (т.е. нулевой аргумент) *это всегда абсолютный путь до исполняемого файла*.

Т.е. если ваша программа лежит тут:

```
/home/iushnik-s-semerki/scripts/my_script.sh
```

То переменная `$0` соответственно будет равна этому значению.

А для остальных:

```
# test.sh
echo \${1}=${1}
echo \${2}=${2}
echo \${3}=${3}
```

```
bash test.sh foo bar "foo bar"
```

```
$1=foo  
$2=bar  
$3=foo bar
```

Можно читать аргументы запуска скрипта не по отдельности, а сразу:

```
# test.sh a b c  
echo $*      # a b c  
echo $@      # a b c
```

В чём отличие? Вернёмся к вопросу в главе 6.

## 2.9. Вопросы существования с точки зрения программиста

А что, если мы попытаемся обратиться к несуществующей переменной? Попробуем:

```
a=5  
echo $a  
echo $b
```

И мы получим...

```
5
```

... просто пустую строку на месте переменной **b**! Никаких вам `NameError: name 'b' is not exists!`

В bash несуществующая переменная является пустой строкой.

Поэтому код ниже абсолютно рабочий:

```
echo $a          # Empty output  
b=$a            # Empty var 'b'  
c="my answer: $a" # my answer:
```

Ещё можно удалить переменную командой `unset`

```
a=5
echo $a      # 5
unset a
echo $a      # nothing
```

А как определять, существует ли переменная? Об этом пойдёт речь в следующей главе.

## Глава III. Ветвления в bash-е или "А что, если..."

### 3.1. Синтаксис

Красиво и понятно. Что ещё можно сказать?

```
if [ $a ]
then
    echo Hello!
fi
```

Несколько логических строчек кода можно расположить на одной физической, применяя разделитель `;`:

```
if [ $a ]; then
    echo Hello!
fi
```

Немного вариативности:

```
if [ $a ]; then
    echo $a
else
    echo ERROR
fi
```

Или так:

```
if [ $a ]; then
    echo $a
else if [ $b ]; then
    echo $b
fi
fi
```

Обратите внимание: каждый раз, открывая условие `if`, необходимо его закрыть `fi`. Дважды использовали `if` — дважды закрыли `fi`.

Однако можно быть проще:

```
if [ $a ]; then
    echo $a
elif [ $b ]; then
    echo $b
fi
```

## 3.2. Чем больше - тем лучше

«Подождите, но я видел такие же условия, но с `[[ ]]`. А в чём отличие?»

Дело в том, что до `bash`-а был `sh`. Но `bash`, в отличие от последнего, имеет множество нововведений. Для обратной совместимости все возможности `shell`-а были сохранены, а новые "фишки" появились в новых скобках - тех самых `[[ ]]`.

Какой вывод можно сделать? Т.к. в курсе ПТП мы рассматриваем исключительно `bash`, и не требуется никакой обратной совместимости со старым интерпретатором, смело используем `[[ ]]`.

Если сомневаетесь, то между `[ ]` и `[[ ]]` выбирайте двойные скобки!

Далее автор будет использовать оба варианта записи.

## 3.2. Социальное дистанционирование

`[ ]` и `[[ ]]` требуют пробелов рядом с собой!

```
if [$a]; then echo foo; fi      # INCORRECT
if [ $a ]; then echo bar; fi   # CORRECT

if [[ $a ]]; then echo foo; fi  # INCORRECT
if [[ $a ]]; then echo foo; fi  # CORRECT
```

## 3.3. Вопросы бытия, часть II

А теперь ответим на вопрос, заданный в предыдущей главе. Проверить переменную на существование очень просто:

```
if [ $a ]; then
    echo "\$a exists; \$a=${a}"
else
    echo "\$a does not exists"
fi
```

Просто и надёжно. Пользуйтесь на здоровье!

## 3.4. Advanced-level ветвлений

Логические операторы и эквивалентность:

В порядке приоритета:

1. **не** = **!**
2. **и** = **&&**
3. **или** = **||**

Пример использования:

```
if [ ! $a ] && ! [ $b ] || [ $c ]; then
    ...
fi
```

**!** можно записать как внутри **[ ]**, так и снаружи, чего не скажешь про **&&** и **||**.

Можно использовать **( )** для изменения приоритета:

```
if ! ([ $a ] && [ $b ]); then
    echo ERROR
fi
```

**==** и **!=**

Несложно догадаться, для чего нужны эти операторы. Записывать внутри скобок:

```
if [[ $a == $b ]]; then
    echo \'a\' is \'b\'
fi
```

```
if [[ $a != $b ]]; then
    echo \'a\' is not \'b\'
fi
```

Тот случай, когда нужно использовать **[[ ]]**. Просто запомните.

## 3.5. Жизни пробелов тоже важны!

Пробелы очень важны! Очень легко потерять пробел и сломать скрипт:

```
# INCORRECT
![ $a ]
!([ $a ] && [ $b ])
[[ $a == $b ]]
[[ $a != $b ]]
```

```
# CORRECT
! [ $a ]
! ([ $a ] && [ $b ])
[[ $a == $b ]]
[[ $a != $b ]]
```

## 3.6. Таинственные минусы

Есть более архаичная запись всего того, что было перечислено:

```
[ $a -a $b ]    # AND
[ $a -o $b ]    # OR
[ $a -eq $b ]   # ==
```

Но тут придётся использовать экранирование ( ):

```
[ ! \( $a -a $b \) ]    # equivalent to "! ([ $a ] && [ $b ])"
```

Забегая вперёд: если `$a` и `$b` являются числами, можно использовать такие операторы:

```
[ $a -lt $b ]    # <
[ $a -ge $b ]    # >=
```

«Зачем это нужно?» Возвращаясь к теме обратной совместимости, такая запись использовалась в shell-e, а в bash перешла по наследству.

На этом автор оставит минусы в покое и далее использоваться они не будут.

## 3.7. =~

Оператор `==~` определяет соответствие строки регулярному выражению.

Синтаксис:

```
if [[ $a =~ ^[+-]?[0-9]+$ ]]; then
    echo '\'$a\' is number!
fi
```

Обратите внимание: **никаких кавычек!**

```
a=123
if [[ $a =~ "[0-9]+$" ]]; then
    echo foo
fi
if [[ $a =~ ^[0-9]+$ ]]; then
    echo bar
fi
```

Вывод будет, как вы можете проверить:

```
bar
```

О регулярных выражениях пойдет речь в 4-ой главе.

## 3.8. Короткая запись условия

Тем, кто честно дочитал до этого момента (да ведь?), будет показан интересный трюк.

Вместо того, чтобы писать

```
if [ ! $a ]; then
    echo ERROR
fi
```

Можно сделать так:

```
[ ! $a ] && echo ERROR
```

Или даже так:

```
[ $a ] && echo $a || echo ERROR
```

Что будет эквивалентно:

```
if [ $a ]; then
    echo $a
else
    echo ERROR
fi
```

Удобно, не правда ли? Но

Однако лучше избегать такой записи условия.

Злоупотребление такой формой приводит к нечитабельности кода.

## Глава IV. Что такое регулярные выражения и с чем их есть?

<https://regex101.com> — регулярные выражения лучше всего писать и тестировать именно здесь. И все примеры лучше проверять здесь

Регулярное выражение, простыми словами — некоторый шаблон, по которому можно определить соответствие строк. Это не строгое сравнение, как в случае с `==`, а "умное". По регулярным выражениям есть целый предмет — Теория Формальных Языков (ТФЯ), кошмар для ИУ9.

Регулярные выражения — очень полезный инструмент, с базовыми основами которого попробуем разобраться в этой главе.

*Единственное, что не могут регулярные выражения — так это [распарсить HTML](#)*

### 4.1. Это база!

Регулярные выражения могут не использовать специальных символов:

```
text
```

Для строки:

```
some text
```

данному регулярному выражению будет соответствовать подстрока:

```
text
```

(проверьте на сайте выше, как это выглядит)

Пример:



```
if [[ "some text" =~ text ]]; then
    echo yes
else
    echo no
fi
```

"some text" - это строка, которую проверяем на соответствие регулярному выражению (на этом месте может быть переменная), и text - как ни странно, **само регулярное выражение**, просто записанное без спец. символов.

Т.к. в строке some text есть подстрока text, результатом работы скрипта будет ответ yes.

## 4.2. Снова [ ]

[ ] - обозначают любой символ внутри. Например:

```
[ab]c
```

Будут соответствовать строки:

```
ac
bc
```

Чтобы не писать весь алфавит, можно использовать диапазоны:

- [A-Z] - заглавные латинские буквы
- [a-z] - строчные латинские буквы
- [A-Z] - латинские буквы (именно в таком порядке!)
- [0-9] - цифры
- [0-f] - так тоже можно, это будут шестнадцатеричные цифры в нижнем регистре

Например:

```
[0-2][0-9]
```

Будут соответствовать строки:

```
00
18
29
```

и т.п.

Чтобы не перебирать все возможные символы, "любой символ" можно задать `.`:

```
..
```

Будут соответствовать строки:

```
ab  
5$
```

## 4.2. Кванторы

Если простым языком, кванторы определяют количество предыдущей строки/символа в подстроке.

- `?` - от 0 до 1

```
[0]?[1-9]
```

Будут соответствовать строки:

```
2  
01
```

- `*` - больше или равно 0

```
[0-9]*
```

Будут соответствовать строки:

```
0  
35434646
```

- `+` - больше 0

```
[0-9]+
```

Будут соответствовать строки:

```
1
1235
```

## 4.3. Экранирование

Чтобы использовать специальный символ как сам символ, используется **экранирование** при помощи `\`. Например, если нам нужно регулярное выражение на проверку расширения файла:

```
.*\.txt
```

Первая точка будет обозначать **любой символ**, а вторая, экранированная, обозначает саму себя, т.е. точку.

## 4.4. Повторения

Если нужно конкретное количество повторений подстроки, используются такие скобки — `{}`:

```
[1-9][0-9]{3}
```

Будут соответствовать строки:

```
1830
2023
```

Или можно записать так:

```
[1-9][0-9]{1-3}
```

Будут соответствовать строки:

```
10
128
1830
```

## 4.5. Группы

Если кванторы нужно применить не к одному символу, можно оборачивать нужную подстроку в *группу* `( )`:

```
(ab)*
```

Будут соответствовать строки:

```
ab
ababab
```

## 4.6. Якоря

Якоря обозначают начало и конец строки:

- `^` - начало строки
- `$` - конец строки

При помощи якорей можно "отбросить" соответствие на подстроки. Например:

```
[[ "file.txt.a" =~ .*\.txt ]]      # True
[[ "file.txt.a" =~ .*\.txt$ ]]    # False
[[ "file.txt" =~ .*\.txt$ ]]      # True
```

## 4.7. Числа!

Итак, наконец мы получили достаточно знаний чтобы написать регулярное выражение для целого числа:

```
^[+-]?[0-9]+$
```

Разберем каждый символ:

- `^` - начало строки, т.е. мы отбрасываем "ab566" значения
- `$` - конец строки, т.е. мы отбрасываем "1223ab" значения
- `[+-]` - любой символ, + или -
- `[+-]?` - обозначет, что "+" или "-" необязательны, но они могут быть в строке, обозначающей число
- `[0-9]` - любая цифра
- `[0-9]+` - цифра должна быть хотя бы одна

Можно попробовать сделать регулярное выражение для дробного числа:

```
^[+-]?[0-9]+(\.[0-9]+)?$
```

Подробнее:

- `[+-]?[0-9]+` - проверка на подстроку-число из предыдущего примера. Обратите внимание, что якоря "переехали", т.к. начало и конец строки теперь в других местах.
- `\.` - точка как символ точки
- `\.[0-9]+` - дробная часть числа
- `(\.[0-9]+)?` - дробная часть числа необязательная

Автор настоятельно рекомендует самостоятельно попробовать написать регулярные выражения для даты, чтобы закрепить материал.

## 4.8. С чем регулярные выражения путать нельзя или маски файлов

Маски файлов похожи на регулярные выражения, но ими не являются. У масок значительно более простой синтаксис, т.к. они предназначены исключительно для поиска файлов. Они имеют очень простой синтаксис:

- `*` - любая последовательность символов, включая пустую ""
- `?` - любой символ

Например, под маску `?a*` подходят имена файлов:

```
2a
bash.md
ca.txt
```

## Глава V. Магия математики

Только теперь можно приступить к математическим операциям. Числа желательно проверять регулярным выражением, т.к. всё в bash-у является строкой и нет отдельного типа данных для чисел.

```
read a b
re_num=^[+-]?[0-9]+$
if ! ([ $a =~ re_num ] && [ $b =~ re_num ]); then
    echo ERROR
else
    ...
fi
```

### 5.1. Школьная арифметика

Bash не поддерживает арифметические операции с дробными числами.

Совсем. И незачем. Bash-у не нужно что-то считать, с этой задачей справится любой классический полноценный язык программирования. Поэтому берём только целые числа.

Для произведения **любых математических** операций, используются двойные круглые скобки: `(( ))`:

```
a=2
(( b = a * a ))
echo $b    # 4

(( b++ ))
echo $b    # 5

(( b *= -1 ))
echo $b    # -5
```

Как Вы можете заметить, **внутри** скобок `$` не обязателен. Однако его нужно использовать для особых переменных, например, для позиционных:

```
# test.sh 5 2
(( res = $1 + $2 ))
echo $res    # 7
```

Какие операции поддерживаются?

- `+`, `-`, `*`, `/` (целочисленное), `%`, `**`
- `+=`, `-=`, `*=` и т.п.
- `++` и `--`

## 5.2. Возврат результата вычисления

Запустив скрипт

```
b=2
echo (( b * 2 ))    # bash: syntax error
```

Вы получите ошибку, т.к. `(( ))` ничего не возвращают.

Чтобы вычислить результат и сразу вернуть его, используется, как можно догадаться, знак доллара перед (так же, как и чтение переменной):

```
b=2
echo $(( b * 2 ))    # 4
```

Злоупотреблять этим символом тоже не стоит:

```
i=0
$(( i += 1 ))      # bash: 1: command not found...
```

Как в случае с чтением переменных, интерпретатор подставляет на место доллара результат вычисления:

```
$(( i += 1 ))
#      |
#      i -> 0
#      |
#      $(( 0 + 1 ))
#      |
#      0 + 1 = 1
#      |
#      $(( 1 ))
#      |
#      $(( 1 )) -> 1
#      |
#      V

1      # bash: 1: command not found...
```

### 5.3. Конкатенация - не сложение!

Для сложения строк и чисел во многих ЯП используется символ `+`:

```
# Python3
a = "foo"
b = "bar"
c = a + b
print(c)  # foobar
```

Но не в bash. Любая попытка подставить *не число* в математические скобки не увенчается успехом.

```
a=foo
b=bar
echo $((a + b))  # 0
```

Правильная запись "сложения" (конкатенации) строк:

```
a=foo
b=bar
echo ${foo}${bar}  # foobar
```

## 5.4. Математические условия

При помощи `(( ))` можно записывать условия, использующие математические операции:

```
if (( a % 2 == 0 )); then
    echo $a - odd
else
    echo $a - even
fi
```

## Глава VI. Вот они, слева направо: `while`, `until`, `for`

### 6.1. `while` и `until`

`while` - самый обычный, но не всегда самый удобный способ задать цикл:

```
while [ $condition ]; do
    ...
done
```

В качестве `condition` может выступать любое выражение - как в ветвлениях:

```
i=0
while [[ $i < 5 ]]; do
    echo -n "$i "
    (( i++ ))
done
# 0 1 2 3 4
```

Редко, но используется другая форма записи - через `until`:

```
i=0
until [[ $i >= 5 ]]; do
    echo -n "$i "
    (( i++ ))
done
# 0 1 2 3 4
```

В чём отличие? Цикл `while` работает, пока значение *ИСТИННО*, в то время как `until` работает, пока значение *ЛОЖНО*.



Конечно, можно не морочить голову и везде писать `while`... Однако если есть такая возможность в языке, то почему бы не использовать? План такой:

- Записывать условие для `while`
- Если в условии есть отрицание всего выражение, например:

```
! ([[ $a ]] && [[ $b ]])
```

в таком случае проверяете, возможно ли избавиться от символа, уменьшив количество операторов. Если же нет, то записываете цикл через `until`. Например:

```
while !([[ $a ]] && [[ $b ]]); do
    ...
done
```

можно трансформировать в

```
until [[ $a ]] && [[ $b ]]; do
    ...
done
```

И такая запись будет короче и приятнее глазу.

## 6.2. Всемогущий `for`

### 6.2.1. Для массивов

Самый типичный пример использования `for` - перебор значений в массиве:

```
for it in array; do
    echo $it
done
```

Например:

```
OLD_IFS=$IFS
IFS=" "

line="i love bash scripts"

for word in $line; do
    echo word
done
```

```
IFS=$OLD_IFS
```

```
# i
# love
# bash
# scripts
```

К массивам вернёмся в 7 главе.

Но у `for` есть ещё несколько удобных конструкций, которые и делают его основным способом задания цикла.

### 6.2.2. Для простых последовательностей

Очень просто записывается `for` для последовательностей:

```
for i in $(seq 1 3); do
    echo -n "$i "
done
# 1 2 3
```

Обратите внимание: границы включены в последовательность!

### 6.2.3. Си-стайл

Если же Вам нужно записать сложный цикл, как в С-подобных языках программирования, то Вам на помощь придут математические скобки:

```
for (( i=4; i>=0; i-- )); do
    echo -n "$i "
done
# 4 3 2 1 0
```

Но не стоит злоупотреблять такой возможностью. Если будет что-то такое:

```
for (( i=0; i<5; i++ )); do
    ...
done
```

можно заменить на более простую запись через `seq`:

```
for i in $(seq 0 4); do
    ...
done
```

```
done
```

Ещё раз обратим внимание: в **seq** обе границы последовательности включительно!

## 6.3. Обработка позиционных параметров

Ранее мы разобрали, что такое позиционные параметры и команду **shift**. Пора применить их вместе:

```
# test.sh foo bar "hello world"

while [ $1 ]; do
    echo $1
    shift
done

# foo
# bar
# hello world
```

Скрипт выводит построчно аргументы при запуске скрипта (за исключением нулевого). Вот так это работает:

```
# /pathtoyourscript/test.sh foo bar hello world
#           ^           ^   ^           ^
#           $0          $1  $2          $3

echo $1      # foo

shift        # ----->
# /pathtoyourscript/test.sh foo bar hello world
#           ^           ^           ^
#           $0          $1          $2

echo $1      # bar

shift        # ----->
# /pathtoyourscript/test.sh foo bar hello world
#           ^           ^           ^
#           $0          $1          $2

echo $1      # hello world

shift        # ----->
# /pathtoyourscript/test.sh foo bar hello world
#           ^
#           $0
```