НИУ ИТМО
Кафедра ВТ

«Системное программное обеспечение»

Тезисы лекций к разделу

«Программирование в Korn Shell»

Дергачев А.М.

Санкт-Петербург, 2014 г.

Назначение интерпретаторов

- *Интерактивное использование* – интерфейс пользователя с операционной системой Unix (интерфейс командной строки)
- *Настройка сеанса* – определение переменных, определение реакции системы на нажатие управляющих клавиш, стартовые командные файлы: /etc/profile, ~/.profile и ~/.kshrc (в лекциях рассматривается **Korn Shell**)
- *Программирование* – использование встроенных команд интерпретатора и команд операционной системы для создания сценариев на языке командного интерпретатора

Версии интерпретаторов

Для определения версии интерпретатора **Korn Shell**, с которым в настоящей момент Вы работаете, можно выполнить следующую последовательность команд:

$ **type ksh**

ksh is /usr/bin/ksh

$ **what /usr/bin/ksh**

/usr/bin/ksh:

*Version M-11/16/**88**i*

SunOS 5.10 Generic 146054-04 May 2011

Простой командный файл

$ **cat > sample.ksh**

date

^D

Варианты запуска файла сценария на выполнение:

1. ksh sample.ksh
2. sample.ksh

В первом случае файл должен быть доступен для чтения.

Во втором случае файл должен быть доступен для чтения и выполнения. Кроме того в переменной PATH должен быть прописан путь в «текущий» каталог:

PATH=$PATH:.

chmod +rx sample.ksh

Комментарии

- знак комментария в сценариях интерпретатора

#!/bin/ksh – особый случай. Используется в первой строке сценария для
указания программы, исполняющей сценарий (sh, bash, awk)

```
date    #   данная команда UNIX возвращает дату и время
df      #   данная команда UNIX сообщает о том, сколько места
        #   доступно в настоящий момент на диске

# Пустые строки или строки содержащие только пробельные символы
# игнорируются. Их можно использовать для улучшения читабельности
# сценариев интерпретатора
```

Использование строки USAGE

```
USAGE="usage: sample.ksh"  # данная строка говорит о том, как запустить
                           # сценарий sample.ksh на выполнение
date       #    данная команда UNIX возвращает дату и время
df         #    данная команда UNIX сообщает о том, сколько места
           #    доступно в настоящий момент на диске

$ sample.ksh
Sat Feb 14 16:04:50 EST 2011
/dev/dsk/W0d0s1   (//node_2707c):          73304 blocks       (N/A)   i-nodes

$ grep USAGE sample.ksh
USAGE="usage: sample.ksh"  # данная строка говорит о том, как запустить
```

Вывод на экран

USAGE=“usage: output.ksh” # Пример организации простого вывода
echo “Здравствуйте товарищи студенты.” # Использование команды echo
print # Вывести пустую строку
Экранирование или цитирование служебных символов: **‘**, **“**, ****
print **–n** “Сегодня “ # -n подавляет вывод символа новой строки
print **–n** “мы рассмотрим:“
При выводе могут использоваться управляющие последовательности:
Print “**\n\t**Типы данных, математические операции и шаблоны”

$ output.ksh
Здравствуйте товарищи студенты.

Сегодня мы рассмотрим:
Типы данных, математические операции и шаблоны

Переменные

Имена переменных могут содержать **буквы**, **цифры** и **подчеркивание** (_)

```
Legal_variable_name  # допустимое имя переменной
v                    # допустимо, но не информативно
illegal variable name  # недопустимое – пробелы в имени переменной
86                   # недопустимое – начинается с цифры
variable86           # допустимое имя переменной
```

Для просмотра списка установленных переменных и их значений можно использовать команду **set**

TOM, **Tom** и **tom** – это три разные переменные

Ввод с клавиатуры

```
USAGE="usage: input.ksh"   # Пример организации простого ввода

print –n "Введите слово или фразу: "
read user_input
print –n "Введите три числа через пробел: "
read first second third
print "\t$user_input, $first.$second.$third"
```

```
$  input.ksh
Введите слово или фразу: Tom Cat
Введите три числа через пробел: 15 02 2011
        Tom Cat, 15.02.2011
```

Присваивание значения переменным

USAGE="usage: assign.ksh" # Пример присваивания значения переменной
n=100 # Четыре варианта присвоения численного значения
let n=100
let "n = 100"
((n = 100))
Print "Значение переменной n равно: $n"
letter="Q"; **word="Cat"**; **phrase="Tom cat"**
print "Letter = $letter;\tWord = $word;\tPhrase = $phrase"
x=$n; print "x = $x"

$ assign.ksh
Значение переменной n равно: 100
Letter = Q; Word = Cat; Phrase = Tom cat
x = 100

Ошибки присваивания

y = 100 # **Пробелы** между именем переменной
title = “Ошибки присваивания” # знаком присваивания и значением
# В результате буде выдано сообщение об ошибке типа:
y: not found
# Ошибки использования **знака $** при работе с переменными:
name=Tom # Значение *Tom* присвоено переменной *name*
cat=name # **Ошибка 1:** Само слово «name», а не значение
# переменной *name* присвоено в качестве значения
#переменной *cat*
$cat=$name # **Ошибка 2:** Знак **$ не нужен** перед переменной *cat*
cat=$name # Правильное присвоение: значение переменной *name*
# присвоено переменной *cat*

Поиск ошибок

Для поиска ошибок можно выполнить файл сценария в режиме отладки:

$ ksh **–x** input.ksh

+ USAGE=“usage: input.ksh”

+ print –n “Введите слово или фразу: “

Введите слово или фразу: **+ read** user_input

Tom Cat

+ print –n “Введите три числа через пробел: “

Введите три числа через пробел: **+ read** first second third

15 02 2011

+ print “\t**$**user_input, **$**first.**$**second.**$**third”

Tom Cat, 15.02.2011

Либо использовать внутри сценария команды **set –x** и **set +x**

Типы данных

String	Если тип переменной специально не был задан при создании, то переменная имеет символьный тип. Т.е. может содержать любые символы: буквы, цифры, знаки пунктуации. При этом с числами можно выполнять арифметические операции.
Integer	Объявление переменой данного типа ускоряет выполнение арифметических операций и дает возможность устанавливать дополнительные атрибуты, например: систему исчисления (base 2), или количество цифр числа (4 цифры). Integer хранятся как 32-битные значения со знаком.
Arrays	По умолчанию тип массива символьный. Можно задать тип integer. Размер массива не задается (512 или 1024 элемента). Массив одномерный.

Тип Strings

```
USAGE="usage: decl_str.ksh"          # Объявление символьной переменной
letter="A"
book="Ksh Programming Tutorial"
numerical_string="911"
phone_message="Позвони мне по номеру 555-1212"
typeset st; st=$letter
print "Примеры символьных значений переменных:"
print $letter, $book, $numerical_string, $phone_message, $st

$ decl_str.ksh
Примеры символьных значений переменных:
A, Ksh Programming Tutorial, 911, Позвони мне по номеру 555-1212, A
```

Тип Integer

```
USAGE="usage: decl_int.ksh"          # Объявление переменной integer
typeset –i y # или integer y=100
y=100; print –n "y = $y, "
y="Integer" # Попытка присвоения символьного значения переменной
                  # типа integer приведет к сообщения об ошибке
x=150          # переменная x символьная, т.к. по другому ее не объвляли
print –n "x = $x, "
x="Tom"     # Успешное присвоение не числового значения переменной x
print "x = $x"

$ decl_int.ksh
y = 100, x = 150, x = Tom
```

Константы

USAGE=“usage: decl_con.ksh” # Объявление констант

typeser –r const_name=4 # или **readonly** const_name=4

print “const_name = $const_name”
print
const_name=5 # Недопустимо переназначение константы

$ decl_con.ksh
const_name = 4

decl_con.ksh[7]: ksh: const_name: is read only

Массивы

```
USAGE=“usage: decl_ary.ksh”        # Объявление массива
animal[0]=“cat”                    # Символьный массив animal
animal[1]=“dog”                    # Не обязательно присваивать значения
animal[6]=“monkey”                 # подряд всем элементам
# Элементы массива со 2-го по 5-й содержат значение «null»
integer test_scores                # Создан массив test_scores типа integer
test_scores[0]=100                 # Присвоено значение первому элементу
print “Введите пороговое значение для формы оценивания «зачет»: “
read test_scores[1]                # Ввод с клавиатуры значения второго элемента
# test_scores[1]=“Шестьдесят” # Недопустимое присвоение
# Альтернативный вариант создания массива с помощью команды set -A
set –A flowers gardenia “bird of paradise” hibiscus
```

Вывод элементов массива

```
USAGE=“usage: pr_ary.ksh” # Вывод элементов массива
# Создание массива flowers и присвоение ему четырех значений:
set –A flowers gardenia “bird of paradise” hibiscus rose
# Вывод значений отдельных элементов массива flowers:
print “Элемент №0 содержит значение: ${flowers[0]}”
cell_number=2
print “Значение элемента №$cell_number: ${flowers[$cell_number]}”
# Вывод значений всех элементов массива flowers:
print “Flowers: ${flowers[*]}” # или print “Flowers: ${flowers[@]}”

$ pr_ary.ksh
Элемент №0 содержит значение: gardenia
Элемент №2 содержит значение: hibiscus
Flowers: gardenia bird of paradise hibiscus rose
```

Ошибки при работе с массивами

```
array[2]=“tulip”          # Правильно
${array[2]}=“tulip”       # Не правильно. Фигурные скобки не нужны

flower=${array[2]}        # Правильно
flower=array[2]           # Не правильно. Пропущены фигурные скобки

array[3]=${array[2]}      # Правильно

print “${array[2]}“       # Правильно
print “array[2]”          # Не правильно. Пропущен знак $ и {}
print “$array[2]”         # Не правильно. Пропущены фигурные скобки
```

$array является синонимом **array[0]**

Математические операции

Оператор	Операция	Пример	Результат
+	Сложение	((y = 7 + 10))	17
-	Вычитание	((y = 10 - 3))	7
*	Умножение	((y = 5 * 6))	30
/	Деление	((y = 37 / 5))	7
%	Остаток от деления	((y = 37 % 5))	2
<<	Сдвиг влево	((y = 2#1011 << 2))	2#101100
>>	Сдвиг вправо	((y = 2#1011 >> 2))	2#10
&	Логическое И	((y = 2#1010 & 2#1100))	2#1000
|	Логическое ИЛИ	((y = 2#1010 | 2#1100))	2#1110
^	Исключающее ИЛИ	((y = 2#1010 ^ 2#1100))	2#1010

Сложение, вычитание, умножение

```
USAGE=“usage: asm.ksh”     # Сложение, вычитание, умножение
integer x=5; integer y=7; integer z        # Задаем начальные значения
((z = x + y)); print –n “x плюс y равно $z; ”
((z = x – y)); print –n “x минус y равно $z; ”
((z = x * y)); print “x умножить на y равно $z.”
# Не обязательно объявлять переменные как integer
print –n “Введите количество баллов, набранных в модуле №5: “; read mod5
print –n “Введите количество баллов, набранных в модуле №6: “; read mod6
print “Вы набрали всего ((mod5 + mod6)) баллов.”
```

```
$ asm.ksh
x плюс y равно 12; x минус y равно -2; x умножить на y равно 35.
Введите количество баллов, набранных в модуле №5: 32
Введите количество баллов, набранных в модуле №6: 28
Вы набрали всего 60 баллов.
```

Деление

```
USAGE="usage: division.ksh"          # Деление

print –n "Сколько метров от Вашего дома до ближайшей станции метро?"
read distance
((kilometers = distance / 1000))
((remaining_meters = distance % 1000))
print "От Вашего дома до ближайшей станции метро $kilometers
километров и $remaining_meters метров."
```

```
$ division.ksh
Сколько метров от Вашего дома до ближайшей станции метро? 5020
От Вашего дома до ближайшей станции метро 5 километров и 20 метров
```

Дробные числа

Рассматриваемый в данном материале **ksh88** не выполняет математические операции с дробными числами

```
x=7.9
y=2.4
((z = x + y))
print “$x + $y = $z”
```

После выполнения приведенного выше сценария на экран будет выведено:

```
7.9 + 2.4 = 9
```

Для выполнения вычислений с плавающей запятой можно использовать утилиты семейства Awk или языки программирования высокого уровня.

Группировка математических операций

Приоритет и правила группировки математических операций в сценариях языка Korn shell аналогичны приоритетам и правилам группировки математических операций.

Пример перевода значения температуры по Фаренгейту в значение температуры по Цельсию:

```
((centigrade = fahrenheit – 32 * 5 / 9))          # Не правильно

((centigrade = ((fahrenheit – 32) * 5) / 9))      # Правильно
```

Распространенные ошибки

1. (z = x + y) # **Не правильно**. Одинарные скобки имеют другое
   # назначение в языке сценариев ksh.
   ((z = x + y)) # Правильно
2. ((z = (x + y) * (a + b)) # **Не правильно**. Не хватает закрывающей скобки **)**
   ((z = (x + y) * (a + b))) # Правильно
3. Диапазон допустимых значений целых чисел со знаком находится в пределах от -2,147,483,648 до +2,147,483,647. По этому после вычисления выражения ((d = 2000000000 * 3)) в переменной d будет находиться число **1705032704, а не 6000000000**.
4. ((var = 5 – 7)); print “$var” # Будет выдано сообщение об ошибке:
   ksh: print: bad option(s) # ksh считает **-2** опцией команды print
   # Для правильного выполнения команды print надо использовать **–R**
   ((var = 5 – 7)); print –R “$var” # **Правильно**

Двоичные, восьмеричные, шестнадцатеричные и др.

```
USAGE="usage: base.ksh" # Целые числа в других системах исчисления
typeset –i x        # определить x, как целое в десятичной системе
typeset –i2 y       # определить y, как целое в двоичной системе
typeset –i8 z       # определить z, как целое в восьмеричной системе
typeset –i16 h      # определить h, как целое в шестнадцатеричной системе
print –n "Введите целое число: "; read x
h=z=y=x # Присвоение значения переменной x переменным h, z и y
print "$x в двоичной $y, в восьмеричной $z,\n в шестнадцатеричной $h"

$ base.ksh
Введите целое число: 125
125 в двоичной 2#1111101, в восьмеричной 8#175,
 в шестнадцатеричной 16#7d"
```

Ограничение на количество цифр

```
USAGE="usage: digits.ksh" # Ограничение выводимого количества цифр
typeset –Z5 salary        # Переменная salary будет выводится всегда
                          # пятью цифрами
print –n "Введите целое число: "; read salary
print "Salary = $salary"
```

$ digits.ksh

Введите целое число: **7154**

*Salary = **07154***

$ digits.ksh

Введите целое число: **123456**

*Salary = **23456***

Шаблоны соответствия

Шаблон	**Соответствие**
?	один любой символ
[char1char2…charN]	один из перечисленных символов
[! char1char2…charN]	один из не перечисленных символов
[char1-charN]	один из указанного диапазона символов
[! char1-charN]	один из символов вне указанного диапазона
*	любое количество любых символов
?(pattern1|pattern2…|patternN)	ноль или одно из указанных выражений
@(pattern1|pattern2…|patternN)	точно одно из указанных выражений
*(pattern1|pattern2…|patternN)	ноль или более из указанных выражений
+(pattern1|pattern2…|patternN)	одно или более из указанных выражений
!(pattern1|pattern2…|patternN)	любое кроме одного из указанных

?(*pattern1*|*pattern2*…|*patternN*)

Шаблон: car?(t)
Соответствие: car, cart

Шаблон: car?([ted])
Соответствие: car, cart, care, card

Шаблон: care?(ful|less|free)
Соответствие: care, careful, careless, carefree

Шаблон: car?(?|??)
Соответствие: Любые **трех**, **четырех** или **пяти** символьные слова, первыми буквами которых являются **car**: car, card, car54

@(*pattern1*|*pattern2*…|*patternN*)

Шаблон:	car@(t)	*(аналогично шаблону car[t])*
Соответствие:	cart	Не соответствие:car
Шаблон:	car@([ted])	
Соответствие:	cart, care, card	Не соответствие:car
Шаблон:	care@(ful|less|free)	
Соответствие:	careful, careless, carefree	Не соответствие:care
Шаблон:	car@(?|??)	
Соответствие:	Любые **четырех** или **пяти** символьные слова, первыми буквами которых являются **car**: card, car54	

*(*pattern1*|*pattern2*...|*patternN*)

Шаблон:	car*(t)
Соответствие:	**car** или строка, начинающаяся с **car** и следующими затем ноль или более символами **t**: car, cart, cartt, carttt
Шаблон:	*([a-z])
Соответствие:	Пустая строка или строка, содержащая только маленькие буквы
Шаблон:	care*(ful|less|ly)
Соответствие:	care, careful, careless, carefully, carelessly
Шаблон:	?([+-])*([0-9]).*([0-9])
Соответствие:	Пустая строка или дробное число, содержащее десятичную точку: +523.632, -7.2 (-7 не подходит)

+(*pattern1*|*pattern2*…|*patternN*)

Шаблон: car+(t)

Соответствие: **car** или строка, начинающаяся с **car** и следующими затем одним или более символами **t**: cart, cartt, carttt

Шаблон: +([a-z])

Соответствие: Любая строка, содержащая только маленькие буквы

Шаблон: care+(ful|less|ly)

Соответствие: careful, careless, carefully, carelessly (care не подходит)

Шаблон: lecture+([0-9])

Соответствие: Любая строка, начинающаяся с lecture и заканчивающаяся числом: lecture5, lecture11 (lecture не подходит)

!(*pattern1*|*pattern2*...|*patternN*)

Не формальное пояснение: *** - @(*pattern1*|*pattern2*...|*patternN*)**

Шаблон: car!(t)
Соответствие: Любая строка, начинающаяся с **car**, за исключением **cart**: car, cars, car54

Шаблон: !(*.bak)
Соответствие: Любая строка, **не** заканчивающаяся на **.bak**: car, 35, QT
Не соответствие: car.bak, 35.bak, QT.bak

Шаблон: car!(*.bak|*.bu|*_1)
Соответствие: Любая строка, начинающаяся на **car** и **не** заканчивающаяся на **.bak**, **.bu** или **_1**: car, car54
Не соответствие: car.bak, car54.bu, carob_1

Сравнение чисел

Форма записи	**Возвращает истину (true), если**
((*number1* == *number2*))	*number1* равно *number2*
((*number1* != *number2*))	*number1* не равно *number2*
((*number1* < *number2*))	*number1* меньше чем *number2*
((*number1* > *number2*))	*number1* больше чем *number2*
((*number1* <= *number2*))	*number1* меньше или равно *number2*
((*number1* >= *number2*))	*number1* больше или равно *number2*

Сравнение строк

Форма записи	**Возвращает истину (true), если**
[[*string1* = *string2*]]	*string1* равно *string2*
[[*string* = *pattern*]]	*string1* подходит под *pattern*
[[*string1* != *string2*]]	*string1* не равно *string2*
[[*string* != *pattern*]]	*string1* не подходит под *pattern*
[[*string1* < *string2*]]	*string1* предшествует *string2*
[[*string1* > *string2*]]	*string1* следует за *string2*
[[-z *string*]]	нулевая длина строки *string*
[[-n *string*]]	не нулевая длина строки *string*

Сравнение чисел с помощью if

```
USAGE="usage: if_num.ksh"       # Проверка чисел в if
print –n "Введите два числа: "
read x y                        # Распространенные ошибки:
if ((x == y))       # Вариант 1 # ((x = y)) присвоить y переменной x
then                            # (x == y)  сообщение «x: not found»
        print "Числа равны."
fi
if test $x –eq $y; then print "Числа равны."; fi          # Вариант 2
if let "$x == $y"; then print "Числа равны."; fi          # Вариант 3
if [ $x –eq $y ]; then print "Числа равны."; fi           # Вариант 4
if [[ $x –eq $y ]]; then print "Числа равны."; fi         # Вариант 5
```

Обязательная фраза then

```
USAGE="usage: if_then.ksh"      # В отличии от языков C или Pascal
print –n "Введите год: "        # не надо помещать блок команд,
read year                       # идущих за then, между BEGIN и
if (( (year % 4) == 0 ))        # END или фигурными скобками {}
then
        print "$year год кратен четырем."
        print "Возможно это високосный год"
        print "или год проведения Олимпийских игр"
fi
if (( (year % 48) == 0 )); then
        :           # пустой или нулевой оператор
fi
```

Не обязательные фразы else и elif

```
USAGE="usage: if_else.ksh"        # Использование фразы else и elif
print –n "Введите цифру: "        # не является обязательным в if
read n
if (( n < 0 ))
then
        print "Отрицательное число"
elif (( n == 0 ))
then
        print "Ноль"
else
        print "Положительное число"
fi
```

Сравнение строк с помощью if

```
USAGE="usage: if_str.ksh"          # Проверка символьных данных в if
print –n "Введите первую строку символов: "
read str1
print –n "Введите вторую строку символов: "
read str2
if [[ $str1 = $str2 ]]          # Вариант 1
then
        print "Строки одинаковые."
fi
if [ "$str1" = "$str2" ] ; then print "Строки одинаковые."; fi   # Вар. 2
if test "$str1" = "$str2"; then print "Строки одинаковые."; fi # Вар. 3
```

Сравнение строк с помощью < и >

```
USAGE=“usage: if_alpha.ksh”     # Проверка порядка следования
print –n “Введите первую строку символов: “
read s1
print –n “Введите вторую строку символов: “
read s2
if [[ $s1 < $s2 ]]
then
        print “Строка $s1 по алфавиту идет раньше строки $s2”
elif [[ $s1 = $s2 ]]
then print “Строки одинаковые.”
else print “Строка $s2 по алфавиту идет раньше строки $s1”
fi
```

Сравнение строк с шаблоном

```
USAGE=“usage: if_pat.ksh”            # Сравнение строк с шаблоном
print –n “Введите строку символов: “
read s
if [[ $s = c* ]]
        then print “Строка $s начинается с символа «c»”
elif [[ $s = *n ]]
        then print “Строка $s заканчивается символом «n»”
elif [[ $s = @(orange|lemon|lime|grapefruit) ]]
        then print “$s – это цитрус”
        else print “Сравнение не помогло.”
fi
```

Ошибки при сравнении строк

```
if (($str1 == $str2))        # Ошибка:
                             #
                             #
if [[$str1 = $str2]]         # Ошибка:
                             #
                             #
if [[ *.bak = $str ]]        # Ошибка:
                             #
```

Попытка сравнения строк способом, аналогичным сравнению чисел.

Отсутствуют пробелы между скобками и именем переменной.

Переменная и шаблон поменяны местами.

```
if [[ $str = What is your name? ]]        # ? - метасимвол
if [[ $str = “What is your name?” ]]      # ? – просто знак вопроса
```

Логическое ИЛИ и логическое И

```
USAGE="usage: boolean.ksh"     # логические операторы || и &&
print –n "Укажите Ваш возраст: "
read age
if ((age < 7)) || ((age > 64))
        then print "Детям и пенсионерам вход бесплатный"
elif ((age >= 7)) && ((age <= 17))
        then print "Школьникам скидка 50%"
elif ((age >= 18)) && ((age <= 21))
        then print "Студентам скидка 30%"
        else print "А вы ребята платите по полной"
fi
```

Оператор CASE

```
USAGE="usage: case.ksh"          # оператор case

print –n "Укажите Ваш любимый фрукт: "
read fruit
case $fruit in
    "orange")              print "Апельсин – это цитрус";;
    "lemon"|"lime")        print "Лимон и лайм это тоже цитрусы";;

   *)                      print –n "Если он маленький, зеленый "
                           print –n "и кислый, то возможно это тоже "
                           print "цитрус";;
esac      # заканчивает оператор case
```

Шаблоны и метасимволы

```
USAGE="usage: casewild.ksh"    # шаблоны и метасимволы в case
print –n "Могу я Вам чем-нибудь помочь?"
read fruit
case $fruit in
    [Yy][Ee][Ss])          print  "Утвердительный ответ";;
    [Mm]*)                 print  "Не вразумительный ответ";;
    "May be")              print  "Это сообщение мы не увидим!";;
    [Nn][Oo])              print  "Отрицательный ответ";;
    +([0-9]))              print  "Целочисленный ответ";;
   *)                      print  "Тут вряд ли чем-либо поможешь";;
esac      # Еще одна возможная ошибка: (“Лишняя левая скобка”)
```

Проверка файлов и каталогов

Ключи проверки	**Возвращает истину (true), если**
-a *object*	объект существует
-f *object*	объект – регулярный файл либо ссылка
-d *object*	объект является каталогом
-c *object*	объект является символьным устройством
-b *object*	объект – блок ориентированное устройство
-p *object*	объект является именованным каналом
-S *object*	объект является сокетом
-L *object*	объект является символьной ссылкой
-k *object*	на объект установлен “sticky bit”
-s *object*	объект не пустой (не нулевой размер)

Примеры проверки

```
USAGE=“usage: obj_type.ksh”                    # Проверка типа объекта
print –n “Введите полное имя объекта: “
read pathname
if [[ ! -a $pathname ]]
        then print “Объекта с таким именем не существует”
elif [[ -L $pathname ]]
        then print “Объект $pathname символьная ссылка”
elif [[ -d $pathname ]]
        then print “Объект $pathname является каталогом”
        else print “Не известный науке объект”
fi
```

Оператор NOT

```
USAGE=“usage: bool_not.ksh”          # Инверсия условия проверки
if [[ ! -d ~/bob ]]                  # Вместо ~ будет подставлено значение
then                                 # переменной HOME
  print “Каталога по имени ~/bob не существует.”
  mkdir ~/bob                        # Создадим его
fi
if [[ -f /tmp/bob && ! –f /tmp/tom ]]  # Если файл /tmp/bob уже
then                                 # создан, а /tmp/tom еще нет, то
  print “подождем еще пару минут.”
  sleep 2            # Пауза в две секунды
fi
```

Проверка прав доступа

Ключи проверки	**Возвращает истину (true), если**
-r *object*	Мне дано право на чтение файла
-w *object*	Мне дано право на запись в файл
-x *object*	Мне дано право на исполнение файла или право поиска в каталоге
-O *object*	Я являюсь владельцем данного объекта
-G *object*	Я принадлежу к группе владельца объекта
-u *object*	На объект установлен set-user-id бит
-g *object*	На объект установлен set-group-id бит

Примеры проверки

```
USAGE=“usage: obj_type.ksh”                    # Проверка прав доступа
print –n “Введите полное имя объекта: “
read pathname
if [[ -r $pathname ]]; then
        print “Вы можете читать”
elif [[ -w $pathname ]]; then
        print “Вы можете писать”
elif [[ -x $pathname ]]; then
        print “Вы можете исполнять”
else print “Вы ничего не можете”
fi
```

Проверка на эквивалентность

Ключи проверки	**Возвращает истину (true), если**
object1 **–nt** *object2*	*object1* создан позже, чем *object2*
object1 **–ot** *object2*	*object1* создан раньше, чем *object2*
object1 **–ef** *object2*	имя *object2* является вторым именем *object1*

```
USEGE="usage: file_age.ksh"                    # Проверка даты создания
if [[ a.out –ot program.c ]]; then
        print "program.c надо откомпилировать"
        cc program.c
else print "Компиляция не требуется"
fi
```

Группировка нескольких проверок

```
USAGE="usage: mixtests.ksh"          # Комбинирование проверок
print –n "Укажите Ваш возраст: "; read age
print –n "Вы опытный пользователь? "; read mature
if ( ((age >= 18)) || ( ((age >= 16)) && [[ $mature = [Yy]* ]] ) ) &&
   [[ (-f UNIX_internals_book) && (-r UNIX_internals_book) ]]
then
   print "В этой книге 844 страницы. Читайте."
   cat UNIX_internals_book        # (если успеете)
else
   print "Вам лучше воздержаться от чтения этой книги!"
   print "Можем предложить Вам что-нибудь из области Windows"
fi
```

Циклы

```
USAGE="usage: loops.ksh"
integer n=1          # До тех пор,
while ((n <= 4))     # пока условие
do                   # верно
    print "$n"; ((n = n + 1))
done
```

```
# Три оператора цикла
# integer n=1    # До тех пор,
# until ((n > 4)) # пока условие
# do             # не станет верным
#     print "$n"; ((n = n + 1))
# done
```

```
for n in 1 2 3 4  # Тело цикла выполняется для каждого элемента
do                # списка
    print $n
done
```

Досрочное завершение цикла

```
USAGE="usage: break.ksh"                          # Оператор break
integer counter=0; integer total=0
while ((counter < 4)); do                         # Условие завершения цикла
   print –n "Введите число или -99 для завершения: "; read score
   if ((score == -99)); then break; fi # Досрочное завершения цикла
   ((total = total + score)); ((counter = counter + 1))
done
if ((counter != 0)); then
   print "Среднее из введенных значений: $((total / counter))"
   else print "Нет данных для вычисления среднего значения"
fi
```

Пропуск части итерации цикла

```
USAGE="usage: cont.ksh"                          # Оператор continue
integer counter=0; integer total=0
while ((counter < 5)); do                        # Условие завершения цикла
   print –n "Введите число: "; read score
   if (( (score < 0) || (score > 100) )) ; then
         print "Досрочное завершение итерации"
         continue          # пропустим вычисление total и counter
   fi
   ((total = total + score)); ((counter = counter + 1))
done
print "Среднее из введенных значений: $((total / counter))"
```

Вечный цикл

```
USAGE="usage: true.ksh"     # Оператор ':', команды true или false
integer total=0
while :                     # или while true или until false
do
    print "\nСумма всех введенных значений: $total"
    print –n "Введите число или -99 для завершения: "; read num
    if ((num == -99))       # Условие завершения цикла
            then break
            else ((total = total + num))
    fi
done
```

Шаблоны в цикле for

```
USAGE=“usage: for_pat.ksh”          # Шаблоны в списке цикла for
for object in *                     # Вывод списка всех файлов
do                                  # текущего каталога
   print “\t$object”
done
for object in *.c *.s                       # Вывод списка всех файлов,
do                                          # содержащих тексты программ
   if [[ (-f $object) && (-w $object) ]]  # на языке «С» или ассемблере
   then print “\t$object”       # и  доступных для изменения
   fi
done
```

Поиск в подкаталогах

```
USAGE="usage: for_pat.ksh"      # Поиск во вложенных каталогах
for object in */*               # Вывод двух уровней подкаталогов,
do                              # начиная с каталога $PWD
    print "\t$object"
done
for object in   *   */*   */*/*      # Вывод объектов, являющихся
do                                    # каталогами, начиная с $PWD
    if [[ -d $object ]]               # и двумя уровнями ниже
    then print "\t$object"
    fi
done
```

При не совпадении с шаблоном

```
for file in st*      # Если в текущем каталоге не существует файлов
do                   # или каталогов, начинающихся с символов ‘st’,
   wc –l $file       # то в список цикла for будет подставлено ‘st*’,
done                 # что в самом операторе for ошибки не вызовет,
# но команда wc выдаст сообщение об ошибке типа:
#                                   st*: No such file or directory
for file in st*
do                             # В данном примере после подстановки
   if [[ -f $file ]]           # значения ‘st*’ в список цикла for
   then wc –l $file            # выполняется проверка на существование
   fi                          # файла с именем ‘st*’
done
```

Break во вложенных циклах

```
USAGE=“usage: break.ksh”
for x in 1 2 3
do
    print –n “\n\t$x –й проход: “
    for y in 10 20 30
    do
        if (( (x * y) >= 50 )); then
            break
        fi
        print –n “$x-$y,  “
    done
done
```

```
# Вывести пары x и y, произведение
# которых меньше чем 50
    # Результаты выполнения:
# 1-й проход: 1-10, 1-20, 1-30,
# 2-й проход: 2-10, 2-20,
# 3-й проход: 3-10,

# break 2 выполнит выход из
# обоих циклов. В результате
# 1-й проход: 1-10, 1-20, 1-30,
# 2-й проход: 2-10, 2-20,
# Конец обоих циклов
```

Пример простого меню

```
USAGE="usage: select.ksh"                    # Пример простого меню

print "Для завершения скрипта введите символ, обозначающий"
print "конец файла (в ОС UNIX это обычно символ <CONTROL>d)"

PS3="Выберите один из пунктов меню:" # PS3 Подсказка меню

select menu_list in English Russian          # Меню из двух элементов
do
        print "OK."
done
```

Select и case

```
USAGE=“usage: select.ksh”                    # Пример меню с case
print “Для завершения скрипта введите символ, обозначающий”
print “конец файла (в ОС UNIX это обычно символ <CONTROL>d)”
PS3=“Выберите один из пунктов меню:” # PS3 Подсказка меню
select menu_list in English Russian          # Меню из двух элементов
do
    case $menu_list in
        English)  print “Thank you.”;;
        Russian) print “Спасибо.”;;
    esac
done
```

Break и переменная REPLY

```
USAGE="usage: select.ksh"                    # Пример меню с break
PS3="Выберите один из пунктов меню (или нажмите <Enter>):"
select menu_list in English Russian exit     # Меню из трех элементов
do                                           # Выбор заносится в REPLY
    case $menu_list in                       # Если там пустая строка,
        English)  print "Thank you.";;       # то select заново выводит
        Russian) print "Спасибо.";;          # меню на экран.
        exit) break;;                        # Выход из меню.
        *) print "Не верный выбор. Будьте внимательны."
    esac
done
```

Метасимволы в меню

```
USAGE="usage: select.ksh"                    # Метасимволы ? и *
PS3="Выберите один из пунктов меню (или нажмите <Enter>):"
select menu_list in Chapter* exit            # Вместо Chapter* будут
do                                           # подставлены имена
    case $menu_list in                       # файлов текущего
        Chapter[123])  print "Ок!";;         # каталога, начинающиеся
        Chapter[456]) print "Outline";;      # со слова Chapter
        exit) break;;                        # Выход из меню.
        *) print "Не верный выбор. Будьте внимательны."
    esac
done
```

Пример вложенного меню

```
USAGE="usage: select.ksh"                    # Вложенные меню
PS3="Выберите команду:"
select cmd in "list files" exit; do
case $cmd in
    "list files") PS3="Уточните выбор:"
        select option in "quick list" "detailed list"; do
        case $option in
            "quick list") ls; break;;        # Выход из подменю
            "detailed list") ls –l; break;;  # Выход из подменю
        esac; done; PS3="Выберите команду:";;
     exit) exit;;                            # Выход из скрипта
esac; done
```

Аргументы командной строки

1. Необходимые для выполнения программы или скрипта параметры могут запрашиваться интерактивно и вводиться с клавиатуры пользователем:

$ myscript.ksh
Введите имя пользователя: bob
Введите номер группы: 600

2. Параметры можно передать в виде **аргументов или ключей командной строки**:

$ myscript.ksh **bob 600** # или: myscript.ksh **–f** bob **–g** 600

Позиционные параметры

Позиционный параметр	**Значение**
$0	Если это скрипт с аргументами, то имя скрипта; если вызов функции, то имя функции; если команда set, то полное имя самого ksh
$1 $2 … ${10} ${n}	Имя первого аргумента командной строки Имя второго аргумента командной строки … Имя десятого аргумента командной строки Имя n-ного аргумента командной строки
$#	Количество аргументов командной строки
$@ или $*	Все позиционные параметры (от $1 до ${n})

Распространенные ошибки

```
$ print ${11}     # Подстановка 11-го позиционного параметра
$ print $11       # Ошибка. Подстановка 1-го параметра, после
                  # которого допишется цифра ‘1’.
x=$1              # Правильно
$1=“Ошибка”       # Значение переменной $1, как имя переменной
1 =“Ошибка”       # Имя переменной, начинающееся с цифры ‘1’.
```

Параметры $@ и $* идентичны при подстановке их значений, а значения “$@” и “$*” различны при выполнении подстановки:

“$@” преобразуется в “$1” “$2” … “$n”

“$*” преобразуется в “$1*s*$2*s*…$n” # где *s* – это первый символ, # присвоенный переменной **IFS**

Присваивание значений

```
USAGE="usage: param.ksh arg1 arg2"    # Присвоение значений
                                      # параметрам $1 и $2
print "Имя командного файла: $0"
print "Значение первого аргумента командной строки: $1"
print "Значение второго аргумента командной строки: $2"
print "Список значение всех аргументов командной строки: $*"

$ param.ksh dog frog
Имя командного файла: param.ksh
Значение первого аргумента командной строки: dog
Значение второго аргумента командной строки: frog
Список значение всех аргументов командной строки: dog frog
```

Количество позиционных параметров

```
USAGE="usage: $0 arg1" # Проверка наличия аргумента командной
if (($# > 1))                  # строки при вызове arg.ksh
then
        print "Слишком много аргументов для команды $0."
        print "$USAGE" # вывод сообщения: usage: arg.ksh arg1
elif (($# == 1))
then
        print "Корректный вызов команды $0."
else
        print "Не указаны аргументы командной строки."
        print "$USAGE" # вывод сообщения: usage: arg.ksh arg1
fi
```

Значения по умолчанию

```
USAGE=“usage: def.ksh” [pathname_of_dir]          # Необязательный
                                                  # аргумент командной строки
# Если указан хотя бы один аргумент командной строки, то
# переменной dir будет присвоено значение параметра $1, иначе,
# переменной dir будет присвоено значение ‘/export/home/bob’

dir=${1:-/export/home/bob}

print “Выполняется проверка: является ли $dir каталогом.”
if [[ ! –d $dir_name ]]
        print “Увы, но это не является каталогом.”
fi
```

Установка позиционных параметров командой set

```
USAGE="usage: set.ksh"                    # Команда set
set dog pig cat  # Присвоить значения параметрам $1, $2 и $3
for animal in "$@"                  #       Вывод на экран:
do                                  #       dog
        print "\t$animal"           #       pig
done                                #       cat
set –s          # Отсортировать параметры по алфавиту
for animal in "$@"                  #       Вывод на экран:
do                                  #       cat
        print "\t$animal"           #       dog
done                                #       pig
set --          # Удалить все позиционные параметры
```

Сдвиг позиционных параметров

USAGE="usage: shift.ksh ar1 ar2 ar3 ar4 ar5" # Команда shift
print "Количество аргументов командной строки: $#"
print "Значение первого аргумента: **$1**\n"
shift 1; print "После выполнения *shift* осталось **$#** параметров" **#4**
print "Теперь значение первого аргумента: **$1**\n"
shift 2; print "После выполнения *shift 2* осталось **$#** параметров" **#2**
print "Теперь значение первого аргумента: **$1**"

	$1	**$2**	**$3**	**$4**	**$5**
Сначала	dog	frog	pig	cat	bird
shift 1	frog	pig	cat	bird	*unset*
shift 2	cat	bird	*unset*	*unset*	*unset*

Параметры $@ и $*

```
USAGE="usage: average.ksh int1 [int2 … intN]"   # Использование $*
integer total=0
if (($# == 0)); then                            # Если не указано хотя бы одного
        print $USAGE;   exit 1                  # аргумента командной строки, то
fi                                              # завершить скрипт с кодом '1'
for num in $*
do
        ((total = total + num))                 # Просуммировать все аргументы
done                                            # командной строки и подсчитать
((average = total / $#))                        # среднее значение
print "Среднее значение равно: $average"
```

“$*” супротив “$@”

```
USAGE=“usage: expand.ksh”                    # Сравнение “$*” и “$@”
set Ksh88 “Korn Shell” “Programming Tutorial”
for element in “$@”; do          # Вывод на экран: Ksh88
        print “\t$element”       #                 Korn Shell
done                             #                 Programming Tutorial
for element in “$*”; do          # Вывод на экран:
        print “\t$element”       # Ksh88 Korn Shell Programming Tutorial
done                             # (print “$@” выведет тоже самое)
IFS=‘ .,’     # Если первый символ переменной IFS – пробел, то
print “$*” # вывод на экран: Ksh88 Korn Shell Programming Tutorial
IFS=‘;.,’     # Если первый символ переменной IFS – ‘;’, то
print “$*” # вывод на экран: Ksh88;Korn Shell;Programming Tutorial
```

${flowers[*]} и ${flowers[@]}

```
USAGE="usage: pr_ary.ksh"       # Сравнение подстановки * и @
                                # при выводе элементов массива
set –A flowers gardenia "bird of paradise" hibiscus rose
IFS=' .,'      # Если первый символ переменной IFS – пробел, то
print ${flowers[*]}         # в обоих случаях вывод на экран будет:
print ${flowers[@]}         # gardenia bird of paradise hibiscus rose
IFS=';.,'      # Если первый символ переменной IFS – ';', то
print ${flowers[@]}         # в данном случае вывод на экран будет
                            # аналогичен двум предыдущим,
print ${flowers[*]}         # а в данном случае вывод на экран будет:
                            # gardenia;bird of paradise;hibiscus;rose
```

Ключи командной строки

```
USAGE="usage: getopts.ksh [-x] [-y]"          # Обработка ключей
while getopts xy arguments; do                # командной строки
case $arguments in
        x) print "Был введен ключ '-x'";;     # Если ввести '+x'
        y) print "Был введен ключ '-y'";;     # то ошибка выдана
esac; done                                    # не будет

$ getopts.ksh  +x
$ getopts.ksh  –x
Был введен ключ '-x'
$ getopts.ksh  –t
getopts.ksh[2]: getopts: t bad option(s)
```

Недопустимое значение ключа

```
USAGE="usage: invalid_key.ksh [-x] [-y]"          # ':' перед списком ключей
while getopts :xy arguments; do                   # значение arguments = '?'
case $ arguments in                               # в OPTARG – имя ключа
        x) print "Был введен ключ '-x'";;
        y) print "Был введен ключ '-y'";;
       \?) print –n "'$OPTARG' – недопустимое значение ключа; "
           print "$USAGE";;
esac; done
```

```
$ invalid.ksh -x –t
Был введен ключ '-x'
't' – недопустимое значение ключа; usage: invalid.ksh [-x] [-y]
```

On/Off ключи

```
USAGE="usage: on_off_key.ksh [+-d] [+-q]"        # Ключи '+' и '-'
while getopts :dq arguments; do
case $arguments in
          d) compile=on;;          # Не надо ставить перед 'd' знак '-'
         +d) compile=off;;         # в операторе case.
          q) verbose=on;;          # Не надо ставить перед 'q' знак '-'
         +q) verbose=off;;         # в операторе case.
         \?) print –n "'$OPTARG' – недопустимое значение ключа; "
             print "$USAGE";;
esac; done
print "compile = $compile;  verbose = $verbose"
```

Аргументы ключей

```
USAGE="usage: arg_key.ksh [-x number] [-y number]"
while getopts  :x:y:  arguments; do        # ':' после ключа в getopts
case $arguments  in                        # требует наличия аргумента
            x) arg_x=$OPTARG # в OPTARG записан аргумент ключа 'x'
               print –n "Ключ '-x' с аргументом $arg_x. ";;
            y) arg_y=$OPTARG # в OPTARG записан аргумент ключа 'y'
               print –n "Ключ '-y' с аргументом $arg_y. ";;
          \?) print –n "'$OPTARG' – недопустимое значение ключа; "
               print "$USAGE";;
esac; done
$ arg_key.ksh  –x1366  –y768
Ключ '-x' с аргументом 1366. Ключ '-y' с аргументом 768.
```

Недопустимое или пропущенное значение аргумента ключа

```
USAGE="usage: miss_arg_key.ksh [-y number]"

while getopts  :y:  arguments; do
case $arguments  in
        y) height=$OPTARG;;
        #  если пользователь забудет указать аргумент ключа, то
        #  переменной arguments будет присвоено значение ':'
        : ) print "Пропущен аргумент ключа $OPTARG";;
        \?) print –n "'$OPTARG' – недопустимое значение ключа; "
            print "$USAGE";;
esac; done
```

Порядок обработки ключей и аргументов командной строки

```
USAGE="usage: mix_arg.ksh [-x num] [-y num] [color1 … colorN]"
# В OPTIND номер текущего аргумента, анализируемого getopts,
while getopts  :x:y:  arguments; do                    # начиная с единицы
case $arguments in
            x) width=$OPTARG;;    # аргумент ключа x
            y) height=$OPTARG;;   # аргумент ключа y
           \?) print –n "'$OPTARG' – недопустимое значение ключа; "
               print "$USAGE";;
esac; done
((positions_occupied_by_switches = OPTIND – 1))
shift $positions_occupied_by_switches  # остались только не ключи
print "Размеры окна: $width x $height; Цвета: $*"
```

Функция с одним аргументом

```
USAGE="usage: one_arg.ksh"          # Пример простой функции.
function sqr                        # Имя функции sqr.
{                                   # Функция будет выполнена
    ((s = $1 * $1))                 # только при обращении к ней.
    print "Квадрат $1 равен $s"
}
# После присвоения переменной USAGE будут последовательно
# выполнены ниже следующие команды:
print –n "Введите целое число: "
read var_integer
sqr $var_integer                    # Вызов функции и передача параметра.
print "Скрипт завершен."            # Продолжение скрипта после функции.
```

Аргументы функции и аргументы командной строки

```
USAGE="usage:  confused.ksh  integer"   # Плохая идея
function sqr { ((s = $1 * $1)); print $s; }   # Параметр $1 не определен,
sqr        # т.к. при вызове функции он не передается

$ confuse.ksh 7
sqr[2]: s = * : syntax error

USAGE="usage:  clearer.ksh  integer"      # Это уже лучше
function sqr {
        ((s = $1 * $1)); print $s
}
sqr $1    # параметр $1 передается  функции при ее вызове
```

Функция с несколькими аргументами

```
USAGE="usage: mul_args.ksh"          # Вызов функции с передачей
function sqr {                       # двух параметров
        ((s = $1 * $1))
        print "$2:  квадрат $1 равен $s"
}
# После присвоения переменной USAGE будут последовательно
# выполнены ниже следующие команды:
integer var_integer
print –n "Введите целое число: "; read var_integer
print –n "Введите свое имя: "; read your_name
sqr $var_integer "$your_name"        # Передача двух параметров
print "Скрипт завершен."   # Продолжение скрипта после функции.
```

Вызов другого скрипта

```
USAGE=“usage: call_scr.ksh”                    # Вызывающий скрипт
pirnt –n “Введите целое число: “
read var_integer
square.ksh $var_integer  # вызов скрипта и передача параметра

USAGE=“usage: square.ksh”                      # Вызываемый скрипт
((s = $1 * $1))
print “Квадрат $1 равен $s”

$ call_scr.ksh
Введите целое число: 6
Квадрат 6 равен 36
```

Возврат результата выполнения функции командой return

```
USAGE="usage: return.ksh"        # Допустимые значения: от 0 до 255
function sqr {
      ((s = $1 * $1))
      return $s                  # Возвращает значение переменной s
}
# После присвоения переменной USAGE будут последовательно
# выполнены ниже следующие команды:
integer var_integer
print –n "Введите целое число: "; read var_integer
sqr $var_integer
returned_value=$?
print "Квадрат $var_integer равен $returned_value"
```

Возврат результата выполнения функции с помощью $(...)

```
USAGE="usage: retprmtr.ksh"    # Возврат значений из функции
function sqr {
        ((s = $1 * $1));
        print "Input -> $1; Output -> $s"
}
print –n "Введите целое число: "; read var_integer
returned_value=$(sqr $var_integer)
print "Функция sqr возвращает значения: $returned_value"

$ retprmtr.ksh
Введите целое число: 6
Функция sqr возвращает значения: Input -> 6; Output -> 36
```

Одинаковые имена глобальных и локальных переменных

```
USAGE="usage: samename.ksh"          # Комментарий
integer x          # глобальная переменная x
function foo {
        integer x; x=3; print "\tЛокальная переменная: $x"; }
x=10; print "Значение глобальной переменной равно $x"
foo
print "Значение глобальной переменной осталось равным $x"

$ samename.ksh
Значение глобальной переменной равно 10
        Локальная переменная: 3
Значение глобальной переменной осталось равным 10
```

Передача массива в качестве параметра в функцию

```
USAGE="usage: passarray.ksh"
function  find_minimum { set –s; print "Минимум: $1"; }
function  gather_input {
      integer counter=0;  typeset –Z3 array
      while read value; do
            array[$counter]=$value;  ((counter = counter + 1))
      done < $1
      find_minimum ${array[*]}
}
print –n "Введите имя файла данных: "; read datafile
gather_input  $datafile              # datafile содержит целые числа
```

Авто подгружаемые функции

$ **mkdir** $HOME/**my_funcs**
$ **vi** $HOME/my_funcs/**sqr**
function sqr { ((s = $1 * $1)); print “Квадрат $1 равен $s”;)) }

USAGE=“usage: callauto.ksh” # Вызов авто подгружаемой функции
autoload sqr # функция sqr определена вне данного скрипта
FPATH=$HOME/my_funcs:/usr/groupfun # Переменная FPATH
должна содержать пути ко всем каталогам,
содержащим авто подгружаемые функции
print –n “Введите целое число: “; read var_integer
answer=$(sqr $var_integer) # Вызов авто подгружаемой функции
print “$answer”

Арсенал ввода/вывода

1. Оператор **read** получает строку ввода и разбирает ее на отдельные элементы в соответствии с символами разделителями, установленными переменной **IFS**. Его работа аналогична функции scanf языка C.
2. Оператор используется **print** для вывода данных (или **echo**).
3. Оператор используется **exec** для управления потоками ввода/вывода, если необходимо использование потоков, отличных от stdin (0), stdout(1), stderr (2). Его работа аналогична функции fopen языка C.
4. Не обязательно открывать файл оператором **exec**, чтобы читать из файла или записывать в него. Можно использовать операторы перенаправления потоков ввода/вывода (**|, <, >**).

Оператор read и переменная REPLY

```
USAGE="usage: read1.ksh"  # Приглашение для ввода с клавиатуры
                          # в операторе read
print –n "Введите любую строку: "
read st                   # ожидание ввода с клавиатуры
print "Вы ввели строку: $st"

read name?"Введите имя пользователя: "
print "$name"

print –n "Введите название города: "
read
print "Значение переменной REPLY: $REPLY"
```

Ввод нескольких значений

```
USAGE=“usage: read_tok.ksh”                # чтение трех значений
print –n “Введите фамилию, имя и отчество: ”
read fam im ot
print “Фамилия: $fam“
print “Имя: $im“
print “Отчество: $ot“

$ read_tok.ksh
Введите фамилию, имя и отчество: Ms. Tucker Marilyn Joyce
Фамилия: Ms.                    # Все входные данные получены,
Имя: Tucker                     # но сохранены не в тех переменных,
Отчество: Marilyn Joyce         # в которых было задумано
```

Игнорирование лишнего ввода

```
USAGE=“usage: ignore_input.ksh”          #          лишние данные
print –n “Введите фамилию, имя и отчество: ”
read fam im ot ignore
print “Фамилия: $fam \nИмя: $im \nОтчество: $ot“
print “Это можно было и не вводить: $innore”

$ read_tok.ksh
Введите фамилию, имя и отчество: Tucker Marilyn Joyce Ms.
Фамилия: Tucker          # Все необходимые входные данные
Имя: Marilyn             # сохранены в соответствующих переменных
Отчество: Joyce          # Излишки ввода можно проигнорировать
Это можно было и не вводить: Ms.
```

Ввод данных разного типа

USAGE="usage: any_type.ksh" # разнородные данные

integer day; integer year
print –n "Введите дату рождения в формате Месяц День Год: "

В ниже следующей оператор read содержит три переменные
для ввода данных. Первая **month - символьная** (по умолчанию).
Две другие, **day и year**, типа **integer**.

read month day year
print "Вы родились в $year году"

Переменная IFS

USAGE=“usage: ifs.ksh” # переназначение разделителей

IFS =“.!?” # изначально содержит **пробельные** символы

print –n “Введите фразы, заканчивающиеся символами «. ! или ?»: “

read s1 s2 s2 junk; print “Первое предложение: $s1”

print “Второе предложение: $s2”print “Третье предложение: $s3”

$ ifs.ksh

Введите несколько предложений: The Tomcat. A Tom? A cat!

Первое предложение: The Tomcat

Второе предложение: A Tom

Третье предложение: A cat

Ввод более одной строки

```
USAGE="usage: input_par.ksh"  # многострочный ввод данных

print "Введите несколько строк  текста. Последним символом "
print "в строке (за исключением последней) должен быть '\'."
read paragraph
print "$paragraph"

print "Введите несколько строк  текста. Последним символом "
print "в строке (за исключением последней) должен быть '\'."
read –r paragraph          # Игнорировать специальное значение '\'
print "$paragraph"
```

Ввод с использованием цикла

```
USAGE=“usage: input_while.ksh”          # цикл для ввода данных

integer running_total=0
integer a_number
print “Введите несколько чисел (по одному в строке). Закончите
print “ввод нажатием комбинации клавишей <CONTROL>d”
while read a_number
do
    ((running_total = running_total + a_number))
done
print “\nСумма чисел равна: $running_total”
```

Чтение файла из командной строки

```
USAGE="usage: input_while.ksh < datafile" # перенаправление
                                          # в командной строке
integer running_total=0
integer a_number
# Ниже следующий цикл суммирует значения чисел, находящихся
# в файле с именем, указанном в командной строке после '<'
while read a_number
do
    ((running_total = running_total + a_number))
done
print "\nСумма чисел равна: $running_total"
```

Чтение файла из скрипта

```
USAGE="usage: input_while.ksh " # перенаправление входного
                                # потока внутри командного
integer running_total=0         # файла (см. фразу done)
integer a_number
# Ниже следующий цикл суммирует значения чисел, находящихся
# в файле с именем scores, указанном в операторе while после '<'
while read a_number
do
    ((running_total = running_total + a_number))
done < scores
print "\nСумма чисел равна: $running_total"
```

Распространенные ошибки

Оператор перенаправления ввода ‘**<**‘ должен находится справа
от ключевого слова **done** оператора while. Не располагайте его в
строке, содержащей оператор read. Это приведет к организации
бесконечного цикла. Например, ниже следующий оператор while
заставит оператор read вводить одну и ту же строку (первую
строку файла file) при каждой итерации цикла.

while read < file # Будет создан бесконечный цикл

Проверьте существует ли файл и доступен ли он для чтения (**-f, -r**)
Используйте read **–r** вместо read (не обработанный режим ввода)
Используйте дополнительную переменную (**ignore**) для
игнорирования (сохранения в ней) не желательного ввода
данных (больше, чем предусмотрено переменных для ввода)

Копанда print

```
USAGE=“usage: print.ksh”                    # использование escape
                                            # последовательностей
print “Это звуковой сигнал:  \a”
print “Удалить символ k в слове backspace: Back\bspace”
# Использование ключ ‘–r’ позволяет оператору print игнорировать
# управляющие последовательности
print –r “Воспринимать \b как обычные символы: Back\bspace”
# Использование ‘--’ позволяет оператору print игнорировать
# специальное значение символа ‘-’.
print -- “-5 + 7 равно +2”    # иначе сообщение print: bad option(s)
print –r ‘Символ \ является специальным символом.’  # или -R
print “Символ \\ является специальным символом.”
```

Метасимволы в строке

USAGE=“usage: wildstr.ksh” # метасимволы в строке ввода

print –n “Введите строку символов: “
read line
print $line

$ wildstr.ksh
Введите строку символов: Слова ***s** заканчиваются на букву ‘s’
Слова lines numbers scores words заканчиваются на букву ‘s’

set –o noglob # отключить режим интерпретации метасимволов
set +o noglob # включить режим интерпретации метасимволов

Перенаправление вывода >

```
USAGE="usage: redirect_out.ksh > out_file"    # перенаправление
                                              # выходного потока
                                              # в командной строке
read a_string           # ввод со стандартного устройства ввода
print "$a_string"       # вывод на стандартное устройство вывода
read a_number           # ввод со стандартного устройства ввода
print "$a_number"       # вывод на стандартное устройство вывода

$ redirect_out.ksh > out_file    # содержимое файла out_file
bob                              # можно просмотреть командой
600                              # cat out_file
```

Перенаправление >>

```
USAGE="usage: append.ksh  filename"   # перенаправление с
                                      # добавлением в файл
integer running_total=0
integer number
# Ниже следующий цикл суммирует значения чисел из файла
# с именем, находящемся в позиционном параметре $1
while read number
do
    ((running_total = running_total + number))
done < $1
# Добавить полученную сумму в конец того же файла
print "\nСумма чисел равна: $running_total" >> $1
```

Перенаправление < и >

```
USAGE=“usage: i_o_redirect.ksh ”          # совместное использование
                                          # перенаправления ввода и
                                          # вывода
print –n “Введите полное имя входного файла: “
read input_file
print –n “Введите полное имя выходного файла: “
read output_file
while read first_token rest_of_line       # зачем rest_of_line ???
do
        print “$first_token”
done < $input_file > $output_file
print “Все готово!”
```

Не именованные каналы

```
$ command1 > /tmp/file        # выходной поток запишется в file
$ command2 < /tmp/file        # file будет направлен во входной
$ rm /tmp/file                # поток команды command2

$ command1 | command2         # не именованный программный
                              # канал или конвейер
$ ls –l > mydir
$ grep ‘Nov 14’ < mydir

$ ls –l | grep ‘Nov 14’

$ grep ‘^dog ‘ animal_journal | sort | uniq |wc -l
```

Перенаправление вывода ошибок

```
USAGE="usage: err_redirect.ksh C_file1 … [C_fileN] 2 > err_file" #
for file in $*; do
    print –n       "$file: "          # вывод имен файлов в std_out
    print –n –u2 "$file: "            # вывод имен файлов в std_err
    cc –c $file                       # компиляция файлов
    if (($? == 0)); then
        print       "нет ошибок"
        print –u2 "нет ошибок"
     else
        print       "Ошибка. Смотри файл сообщений"
    fi
done
```

Открыть файл с помощью exec

```
USAGE="usage: openfile.ksh"   # Использование оператора exec
exec 8< UNIX_internals_book   # Открыть именованный поток ввода
# Открытый файл получает синоним в виде дескриптора файла 8
read –u8 first_line < UNIX_internals_book      # Первые три строки
read –u8 second_line < UNIX_internals_book     # книги Unix internals
read –u8 third_line < UNIX_internals_book      # будут записаны в
print "$first_line $second_line $ therd_line"  # соответствующие
                                               # переменные.
read first_line < UNIX_internals_book          # Во все переменные
read second_line < UNIX_internals_book         # будет записана
read third_line < UNIX_internals_book          # первая строка
print "$first_line $second_line $ therd_line"  # книги Unix internals
```

Закрыть файл

```
USAGE=“usage: closfile.ksh”      # Использование оператора exec
for object in *; do              # Список файлов текущего каталога
    if [[ -f $object ]]          # Работаем только с регулярными файлами
    then
        print –n “$object: “
        exec 3< $object          # открыть файл $object для чтения
        read –u3 first_line      # читать первую строку этого файла
        read –u3 second_line     # читать вторую строку этого файла
        print “$second_line”     # вывести вторую строку на stdout
        exec 3<&-                # закрыть файл с дескриптором 3
    fi
done
```

Работа с несколькими файлами

```
USAGE="usage: exec_ex.ksh file1 file2"  # Первая половина скрипта
if (($# != 2)); then                    # Если аргументы командной строки
        print "$USAGE";  exit 1        # не введены, то завершить скрипт.
    elif [[ (-f $1) && (-f $2) && (-r $1) && (-r $2) ]]; then
        exec 3< $1                      # открыть $1 для ввода
        exec 4< $2                      # открыть $2 для ввода
        exec 5> match                   # открыть match для вывода
        exec 6> nomatch                 # открыть nomatch для вывода
    else                                # Если пользователь ввел неверные
        print "$USAGE"; exit 1          # аргументы, то завершить скрипт.
fi
```

Продолжение скрипта

```
while read –u3 lineA                    # Прочитать строку из файла $1
do
    read –u4 lineB                      # Прочитать строку из файла $2
    if [ “$lineA” = “$lineB” ]          # Сравнить прочитанные строки
    then
        print –u5 “$lineA”                      # Если строки совпадают, то
    else                                        # записать строку match;
        print –u6 “$lineA; $lineB”              # если нет, то обе строки
    fi                                          # записать в файл nomatch.
done
print “Сравнение файлов выполнено. См. файлы match и nomatch!”
```

Подстановка вывода команды

```
USAGE="usage: cmd_out.ksh file word1 word2" # Конструкция $(…)
date_var=$(date); print "Сегодня $date_var"
line_containing_pattern=$(grep "$2" $1)
print "\nСлово $2 встретилось в файле $1 в следующих строчках:"
print "$line_containing_pattern"
if (( $(grep –c "$2" $1) > $(grep –c "$3" $1) )); then
    print "\n$2 встретилось в файле $1 больше раз чем слово $3."
elif (( $(grep –c "$2" $1) )) < $(grep –c "$3" $1) )); then
    print "\n$3 встретилось в файле $1 больше раз чем слово $2."
else
    print "\n$3 и $2 встретилось в одних и тех же строках файла $1
fi
```

«Документ здесь»

USAGE="usage: here_doc.ksh file replace_this with_this"

Использование конструкции типа: ***command*** *arguments* **<< *word***

ex - $1 **<<EOD** # *instruction_1_to_**command***

%s/$2/$3/g # . . .

wq # *instruction_1_to_**command***

EOD # ***word***

В данном скрипте запускает текстовый редактор ex, которому

в качестве аргумента командной строки передается имя файла

После запуска редактор ex выполняет две команды, которые

находятся между строкой **ex - $1 <<EOD** и строкой **EOD**. При этом

слово EOD должно находиться **в начале строки**.

Обработка строк

Ниже перечислены наиболее распространенные операции со строками, производимые в сценариях, написанных для ksh:

1. Преобразование регистра символов (маленькие, большие)
2. Начальные и конечные пробелы (выравнивание по границе)
3. Конкатенация (соединение) нескольких строк в одну
4. Работа с частью строки (вырезание или вставка подстроки)
5. Преобразование строки в слова или фразы (разделители в IFS)
6. Сравнение одной строки с другой строкой
7. Сравнение строки с заданным значением или шаблоном
8. Присвоение нового значения символьной переменной

Верхний и нижний регистры

```
USAGE="usage: U_and_lc.ksh"          # Преобразование маленьких
                                     # букв в большие или больших
print –n "Ввыдите строку символов: "  # в маленькие
read a_string
backup_string=$a_string
print "\nИсходное значение: $a_string"
typeset –u a_string        # установить атрибут uppercase
print "UPPERCASE: $a_string; Case sensitive: $backup_string"
typeset –l a_string        # lovercase установить, uppercase сбросить
print "lowercase: $a_string ; Case sensitive: $backup_string"
typeset +l a_string        # сбросить атрибут lovercase
print "Still in lowercase: $a_string ; Case sensitive: $backup_string"
```

typeset –Ln (-LZ, -R, -RZ)

```
USAGE="usage: leftjust.ksh"       # Выравнивание по левой границе
typeset –L10 fam_name; typeset –L8 im_name; typeset –L15 ot_name
while read fam_name im_name ot_name          # Вывод слов в три
do                                           # колонки шириной
    print "$fam_name$im_name$ot_name"        # 10, 8 и 15 символов
done < file_of_names                         # соответственно
typeset –L1 im_initial
print –n "\nВведите имя: "; read im_name; im_initial=$im_name
print "Первая буква имени:  $im_initial"
typeset –LZ im_initial     # отбросить начальные нули при выводе
print –n "Введите дату в числовом формате (дд мм гг): "; read d m y
print "$d-$m-$y"           # ввод: 01 05 11, вывод: 1-5-11
```

Конкатенация строк

```
USAGE=“usage: conc.ksh”                    # Соединение нескольких
read fam?”Введите фамилию: “               # строк в одну
read im?”Введите имя: “
read ot?”Введите отчество: “
complete_name=“Ms. $fam $im $ot”
print “Ваше полное имя -- $complete_name”

$ conc.ksh                                 # Данный скрипт формирует
Введите фамилию: Tucker                    # полное имя с префиксом
Введите имя: Marilyn                       # Ms.
Введите отчество: Joyce
Ваше полное имя -- Ms. Marilyn Joyce Tucker
```

Конкатенация строк

```
USAGE="usage: conc.ksh"          # Соединение константы .backup
name_of_directory='/exam'        # значения переменной $file
ful_name_of_directory=$HOME$name_of_directory
cd $ful_name_of_directory; print "$ful_name_of_directory"
for file in !(*.backup)                  # не создавать копии файлов
do                                       # с суффиксом .backup
    if [[ -f $file ]]; then  name_of_backup_file=$file.backup
        if cp $file $name_of_backup_file; then
            print "$file скопирован в $name_of_backup_file"
        fi
    fi
done
```

Оператор подстроки # (левый)

```
USAGE="usage: substr_left.ksh"          # Оператор подстроки #
animal="tiger"
print "Полная строка -- $animal"
print "Полная строка -- ${animal}"
print "Удалим символ 't' --- ${animal#t}"
print "Удалим символы 'ti' --- ${animal#ti}"
abbrev="tig"
print "Удалим символы '$abbrev' --- ${animal#$abbrev}"
print "Удалим первые два символа --- ${animal#??}"
print "Удалим начало по 'e' включительно --- ${animal#*e}\n"
cats="lions and tigers"; print "Удалить первое слово --- ${cats#* }"
print "Удалить первые два слова --- ${cats#* * }"
```

Оператор подстроки % (правый)

```
USAGE="usage: substr_right.ksh"          # Оператор подстроки %
animal="tiger"; abbrev="ger"
print "Полная строка -- ${animal}"
print "Удалим символ 'r' --- ${animal%r}"
print "Удалим символы 'er' --- ${animal%er}"
print "Удалим символы '$abbrev' --- ${animal%$abbrev}"
print "Удалить символы 'tige' не получится --- ${animal%tige}"
print "Удалим последние два символа --- ${animal%??}"
print "Удалим символы, начиная с 'g' --- ${animal%g*}\n"
cats="lions and tigers"; print "Удалить последнее слово -- ${cats%
*}"
print "Удалить последние два слова --- ${cats% * *}"
```

Применение %

```
USAGE=“usage: rename.ksh”      # Замена суффикса .shd на .shadow
for old_file_name in *.shd; do
    new_file_name=${old_file_name%.shd}.shadow
    mv $old_file_name $new_file_name
    if (($? == 0))
    then
        print “Файл $old_file_name переименован в $new_file_name”
    else
        print “Переименование файла $old_file_name невозможно”
    fi
done
```

или

USAGE="usage: remove.ksh" # Отличие оператора **#** от **##**

simple="abcdabcd"

print "Полная строка -- ${simple}"

print "Удалим символы 'ab' --- ${animal**#ab**}" # без метасимволов

print "Удалим символы 'ab' --- ${animal**##ab**}" # результат тот же **##*ab**

print "Удалим символы '*ab' --- ${animal**#*ab**}" # результат **cdabcd**

print "Удалим символы '*ab' --- ${animal**##*ab**}" # результат **cd ##*<пробел>**

cats="lions and tigers"; print "\nПолная строка: ${cats}"

print "Удалить первое слово --- ${cats**#*** }" # результат **and tigers**

print "Удалить до последнего слова -- ${cats**##*** }" # результат **tigers**

% или %%

USAGE="usage: remove.ksh" # Отличие оператора **%** от **%%**

cats="lions and tigers"

print "Полная строка -- ${cats}"

print "Удалим символы 'ab' --- ${animal**% ***}" # **lions and**

print "Удалим символы 'ab' --- ${animal**%% ***}" # **lions**

oz="lions and tigers and bears"; print "\nПолная строка: ${oz}"

print "Удалить от последнего 'and*' до конца строки -- ${oz **%and*** }"

результат --- **lions and tigers**

print "Удалить от первого 'and*' до конца строки -- ${oz**%%and*** }"

результат --- **lions**

Удаление из середины строки

```
USAGE="usage: remove.ksh"                    # Работа с подстрокой
oz="lions and tigers and bears"; print "\nПолная строка: ${oz}"
words_to_remove=" and tigers"
remaining_left=${oz%$words_to_remove* }
remaining_right=${oz#*$words_to_remove}
safer_oz="$remaining_left$remaining_right"
print –n "Строка без удаленных из середины слов выглядит так: "
print "$safer_oz"
new_text=" and tofu"
new_age_oz="$remaining_left$new_text$remaining_right"
print "Теперь строка выглядит так: $new_age_oz"
```

Длина строки ${#...}

```
USAGE="usage: remove.ksh"      # Для вычисления длины строки
                               # можно использовать следующий
                               # синтаксис:
                               #         lenth=${#string}
IFS=""   #        Установка переменной IFS в нулевое значение
         #        на даст проигнорировать начальные пробелы
         #        при вводе строки
while read –r line             # Пример ввода:
do                             # <пробел>Tom cat.
    print "${#line}\t$line"    # Вывод:
done                           # 9       Tom cat.
```

Значения по умолчанию

str1=“Rain”; str2=“”

r1=${str1:-”Dry”} # Присваивает значение “Rain” переменной r1;

значение переменной str1 не изменяется.

r2=${str2:-”Wet”} # Присваивает значение “Wet” переменной r2;

значение переменной str2 не изменяется.

операция	**присвоение значения**	**по умолчанию**
var=**${**string**:-**expr**}**	var=${string}	var=expr
var=**${**string**:=**expr**}**	var=expr	string=expr; var=expr
var=**${**string**:+**expr**}**	var=${string}	var становится null
var=**${**string**:?**expr**}**	var=${string}	вывод expr на stderr