

13

Рафи Кази ар

НКАбд-03-24

1032238132@pfur.ru

Задание 1: Командный файл с getopt и grep Это задание учит создавать скрипты с обработкой ключей командной строки, как это делают стандартные утилиты.

Листинг скрипта (search.sh):

```
#!/bin/bash

# Значения по умолчанию
inputfile=""
outputfile=""
pattern=""
case_sensitive=0
line_numbers=0

# Обработка ключей с помощью getopt
# Опция 'C' и 'n' не требуют аргумента, поэтому после них
нет двоеточия
# Для длинных опций мы используем флаги. Стандартный
getopt не поддерживает
# длинные опции (--inputfile), поэтому эмулируем их через
короткие (-i, -o, -p).
while getopt "i:o:p:Cn" opt; do
case $opt in
i) inputfile="$OPTARG" ;; # Ключ -i соответствует --inputfile
```

```

o) outfile="$OPTARG" ;; # Ключ -o соответствует --
outfile
p) pattern="$OPTARG" ;; # Ключ -p соответствует --pattern
C) case_sensitive=1 ;; # Включить чувствительность к
реестру
n) line_numbers=1 ;; # Включить вывод номеров строк
\?) echo "Неверный ключ: -$OPTARG" >&2
exit 1 ;;
esac
done

```

```

# Проверка на обязательные параметры
if [[ -z "$inputfile" || -z "$pattern" ]]; then
echo "Использование: $0 -i inputfile -p pattern [-o outfile]
[-C] [-n]"
exit 1
fi

```

```

# Проверка существования входного файла
if [[ ! -f "$inputfile" ]]; then
echo "Файл '$inputfile' не найден."
exit 1
fi

```

```

# Построение команды grep
grep_cmd="grep"
if [[ $case_sensitive -eq 1 ]]; then
grep_cmd="$grep_cmd" # По умолчанию grep чувствителен
к регистру, ключ -i отключает это.
# В задании сказано: -C — различать большие и малые
буквы.
# Это поведение grep по умолчанию, поэтому ничего делать
не нужно.
else
grep_cmd="$grep_cmd -i" # Ключ -i: игнорировать регистр
fi
if [[ $line_numbers -eq 1 ]]; then
grep_cmd="$grep_cmd -n"
fi

```

```

# Выполнение поиска

```

```
# Если указан выходной файл, перенаправляем вывод в него,  
иначе выводим на экран  
if [[ -n "$outputfile" ]]; then  
$grep_cmd "$pattern" "$inputfile" > "$outputfile"  
echo "Результаты поиска сохранены в файл: $outputfile"  
else  
$grep_cmd "$pattern" "$inputfile"  
fi
```

Пример использования:

```
# Поиск без учета регистра (по умолчанию), вывод на экран  
./search.sh -i myfile.txt -p "hello world"
```

```
# Поиск с учетом регистра (-C), с номерами строк (-n) и  
сохранением в файл
```

```

[& krafia@pop-os ]-----[
└─ Documents/arcitur
-> ls
011-lab_shell_prog_2.pdf a.

[& krafia@pop-os ]-----[
└─ Documents/arcitur
-> nano search.sh

[& krafia@pop-os ]-----[
└─ Documents/arcitur
-> sudo chmod +x search.sh

[& krafia@pop-os ]-----[
└─ Documents/arcitur
-> echo "My name is Rafi" >

[& krafia@pop-os ]-----[
└─ Documents/arcitur
-> ./search.sh -i newfile -
My name is Rafi

[& krafia@pop-os ]-----[
└─ Documents/arcitur
-> ./search.sh -i newfile -
Результаты поиска сохранены

[& krafia@pop-os ]-----[
└─ Documents/arcitur
-> cat result.txt
1:My name is Rafi

[& krafia@pop-os ]-----[
└─ Documents/arcitur
->

```

```
./search.sh -i myfile.txt -p "Hello World" -C -n -o result.txt
```

Задание 2: Программа на Си и анализ кода возврата

Это задание демонстрирует, как оболочка и программы взаимодействуют через коды завершения.

Листинг программы на Си (check_number.c):

```

#include <stdio.h>
#include <stdlib.h> // Для функции exit()

int main() {
    int number;
    printf("Введите число: ");
    scanf("%d", &number);

```

```

if (number > 0) {
printf("Число положительное.\n");
exit(1); // Код возврата 1 для положительного числа
} else if (number < 0) {
printf("Число отрицательное.\n");
exit(2); // Код возврата 2 для отрицательного числа
} else {
printf("Число равно нулю.\n");
exit(0); // Код возврата 0 обычно означает успешное
завершение
}
}

```

Компиляция:

```
gcc -o check_number check_number.c
```

Листинг командного файла (number_script.sh):

```

#!/bin/bash

# Запускаем скомпилированную программу
./check_number

# Анализируем код возврата последней команды через
специальную переменную $?
case $? in
0)
echo "Было введено число 0." ;;
1)
echo "Было введено положительное число." ;;
2)
echo "Было введено отрицательное число." ;;
*)
echo "Программа завершилась с неизвестным кодом." ;;
Esac

```

```
[krafi@pop-os ~]$ nano check_number.c
[krafi@pop-os ~]$ gcc -o check_number check_number.c
[krafi@pop-os ~]$ ./check_number
Введите число: 5
Число положительное.
[krafi@pop-os ~]$ echo $?
1
[krafi@pop-os ~]$
```

Задание 3: Создание и удаление файлов Это задание учит работе с аргументами командной строки и циклами.

Листинг скрипта (file_manager.sh):

```
#!/bin/bash
```

```
# Функция для создания файлов
create_files() {
    local num_files=$1
    for ((i=1; i<=num_files; i++)); do
        touch "${i}.tmp"
    done
    echo "Создано $num_files файлов."
}
```

```
# Функция для удаления файлов
delete_files() {
    # Удаляем все файлы, подходящие под шаблон '*.tmp'
    # Использование команды rm с шаблоном может быть
    # опасным,
    # но в рамках этого задания допустимо.
    rm -f *.tmp
}
```

```
echo "Все .tmp файлы в текущей директории удалены."  
}
```

```
# Анализ аргументов командной строки  
if [[ $# -eq 0 ]]; then  
echo "Использование:"  
echo " $0 create <N> - создать N файлов"  
echo " $0 delete - удалить все .tmp файлы"  
exit 1  
fi
```

```
case $1 in  
create)  
if [[ $# -ne 2 ]]; then  
echo "Укажите число файлов для создания."  
exit 1  
fi
```

```
# Проверяем, что второй аргумент - число  
if [[ ! $2 =~ ^[0-9]+$ ]]; then  
echo "Второй аргумент должен быть целым числом."  
exit 1  
fi
```

```
create_files "$2"
```

```
;;  
delete)  
delete_files
```

```
;;  
*)  
echo "Неизвестная команда: $1"  
exit 1
```

```
;;  
esac
```

Пример использования:

`./file_manager.sh create 5` # Создаст файлы 1.tmp, 2.tmp, 3.tmp, 4.tmp, 5.tmp

`./file_manager.sh delete` # Удалит все .tmp файлы

```
[& krafi@pop-os ]-->[🕒 03:58 PM]--> www.krafi.info
└─> Documents/arciture rudn/13 via C v11.4.0-gcc
-> nano file_manager.sh

[& krafi@pop-os ]-->[🕒 04:00 PM]--> www.krafi.info
└─> Documents/arciture rudn/13 via C v11.4.0-gcc took 22s
-> sudo chmod +x file_manager.sh

[& krafi@pop-os ]-->[🕒 04:00 PM]--> www.krafi.info
└─> Documents/arciture rudn/13 via C v11.4.0-gcc
-> ./file_manager.sh create 5
Создано 5 файлов.

[& krafi@pop-os ]-->[🕒 04:01 PM]--> www.krafi.info
└─> Documents/arciture rudn/13 via C v11.4.0-gcc
-> ls
011-lab_shell_prog_2.pdf 3.tmp a.docx check_number.c result.txt
1.tmp 4.tmp a.md file_manager.sh search.sh
2.tmp 5.tmp check_number newfile

[& krafi@pop-os ]-->[🕒 04:01 PM]--> www.krafi.info
└─> Documents/arciture rudn/13 via C v11.4.0-gcc
-> ./file_manager.sh delete
Все .tmp файлы в текущей директории удалены.

[& krafi@pop-os ]-->[🕒 04:01 PM]--> www.krafi.info
└─> Documents/arciture rudn/13 via C v11.4.0-gcc
-> ls
011-lab_shell_prog_2.pdf a.md check_number.c newfile search.sh
a.docx check_number file_manager.sh result.txt

[& krafi@pop-os ]-->[🕒 04:01 PM]--> www.krafi.info
└─> Documents/arciture rudn/13 via C v11.4.0-gcc
->
```

Задание 4: Архивация недавно измененных файлов

Это задание учит комбинировать команды (find, tar) для решения практических задач.

Листинг скрипта (archive_recent.sh):

```
#!/bin/bash
```

```
# Проверяем, указана ли директория
if [[ $# -eq 0 ]]; then
echo "Использование: $0 <директория>"
exit 1
fi
```



```

target_dir="$1"
archive_name="backup_$(date +%Y-%m-%d).tar.gz"

# Проверяем существование директории
if [[ ! -d "$target_dir" ]]; then
echo "Ошибка: Директория '$target_dir' не найдена."
exit 1
fi

# Переходим в целевую директорию, чтобы пути в архиве
были относительными
cd "$target_dir" || exit

# Находим все файлы, измененные за последние 7 дней,
и архивируем их.
# Ключ -print0 и -0 для tar позволяют работать с файлами,
содержащими пробелы.
find . -type f -mtime -7 -print0 | tar -czf "../$archive_name" --null
-T -

echo "Архив создан: $(pwd)/../$archive_name"

```

Пример использования:

```
./archive_recent.sh /path/to/your/directory
```

```
[krafi@pop-os ~]$ nano archive_recent.sh
[04:01 PM] www.krafi.info
Documents/arciture rudn/13 via C v11.4.0-gcc
[04:02 PM] www.krafi.info
Documents/arciture rudn/13 via C v11.4.0-gcc took 15s
[04:02 PM] www.krafi.info
Documents/arciture rudn/13 via C v11.4.0-gcc took 4s
sudo chmod +x archive_recent.sh
[04:02 PM] www.krafi.info
Documents/arciture rudn/13 via C v11.4.0-gcc
./archive_recent.sh
Архив создан: /home/krafi/Documents/arciture rudn/13/../../backup_2025-08-27.tar.gz
[04:02 PM] www.krafi.info
Documents/arciture rudn/13 via C v11.4.0-gcc
ls
011-lab_shell_prog_2.pdf a.md check_number file_manager.sh result.txt
a.docx archive_recent.sh check_number.c newfile search.sh
[04:02 PM] www.krafi.info
Documents/arciture rudn/13 via C v11.4.0-gcc
ls ../
'003-lab_markdown (1).pdf' 1 11 13 3 5 7 9
003-lab_markdown.pdf 10 12 2 4 6 8 backup_2025-08-27.tar.gz
[04:02 PM] www.krafi.info
Documents/arciture rudn/13 via C v11.4.0-gcc
```

Ответы на контрольные вопросы

1. **Каково предназначение команды getopts?**
getopts — это встроенная команда оболочки (shell) для синтаксического разбора (парсинга) аргументов и ключей командной строки. Она упрощает обработку флагов (например, -a, -f filename), обеспечивает стандартное поведение и избегает ошибок при ручном разборе через \$1, \$2, shift.
2. **Какое отношение метасимволы имеют к генерации имён файлов?** Метасимволы (или шаблоны, wildcards) *, ?, [] используются

для генерации списка имен файлов, которые соответствуют заданному шаблону. Этот процесс называется «глоббинг» (globbing). Например, команда `rm *.tmp` сначала генерирует список всех файлов, оканчивающихся на `.tmp`, а затем передает этот список команде `rm` для удаления.

3. **Какие операторы управления действиями вы знаете?**

- **Условные операторы:** `if-then-elif-else-fi`, `case-in-esac`
- **Циклы:** `for-in-do-done`, `while-do-done`, `until-do-done`
- **Логические операторы для управления выполнением команд:**
 - `&&` (И): Выполняет следующую команду, только если предыдущая завершилась успешно (код возврата 0). `cmd1 && cmd2`
 - `||` (ИЛИ): Выполняет следующую команду, только если предыдущая завершилась неудачно (код возврата не 0). `cmd1 || cmd2`

4. **Какие операторы используются для прерывания цикла?**

- `break` — немедленно прерывает выполнение цикла и передает управление командам после `done`.
- `break N` — прерывает выполнение `N` вложенных циклов.
- `continue` — прерывает текущую итерацию цикла и переходит к следующей.
- `exit` — завершает весь скрипт полностью.

5. **Для чего нужны команды `false` и `true`?**
Это команды, которые всегда возвращают определенный код завершения.

- `true` — всегда возвращает код успеха (0). Часто используется для создания бесконечных циклов (`while true; do ...`).
- `false` — всегда возвращает код ошибки (не 0). Может использоваться для принудительного перехода к ветке `||` или для заглушки.

6. **Что означает строка `if test -f man$s/$i.$s`?**

Это проверка условия с помощью команды `test` (синоним `[]`).

- `-f` — проверяет, существует ли файл и является ли он обычным файлом (regular file), а не каталогом или устройством.
- `man$s/$i.$s` — это путь к файлу. Здесь `$s` и `$i` — переменные. Например, если `$s=1` и `$i=printf`, то проверяется существование файла `man1/printf.1`.
- Эквивалентная запись в современном синтаксисе: `if [[-f "man${s}/${i}.${s}"]]`

7. **Объясните различия между конструкциями `while` и `until`.**

- **`while`** — выполняет блок команд **пока** условие (код завершения последней команды в условии) истинно (равно 0). `while condition; do commands; done`
- **`until`** — выполняет блок команд **до тех пор, пока** условие не станет истинным (т.е. пока условие ложно — возвращает не 0). `until condition; do commands; done` **Проще говоря:** `while` работает "пока есть успех", а `until` работает "пока есть неудача".