

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ПО ВЫПОЛНЕНИЮ И ОФОРМЛЕНИЮ ЛАБОРАТОРНЫХ РАБОТ ПО ДИСЦИПЛИНЕ «ОПЕРАЦИОННЫЕ СИСТЕМЫ»

2 СЕМЕСТР

BASH-СКРИПТЫ

Целью лабораторной работы является получение практических навыков по написанию Bash-скриптов для ОС Linux.

Задачи:

1. Самостоятельно изучить синтаксис и важнейшие структуры Bash-скриптов.
2. Научиться применять Bash-скрипты для автоматизации тестирования программ.
3. Закрепить полученные в ходе выполнения лабораторной работы навыки

КРАТКАЯ ТЕОРИЯ


Bash (от англ. Bourne again shell, каламбур «Born again» shell — «возрождённый» shell) — усовершенствованная и модернизированная вариация командной оболочки Bourne shell. Одна из наиболее популярных современных разновидностей командной оболочки UNIX. Особенно популярна в среде Linux, где она часто используется в качестве предустановленной командной оболочки.

Представляет собой **командный процессор**, работающий, как правило, в интерактивном режиме в текстовом окне. Bash также может читать команды из файла, который называется скриптом (или сценарием). Как и все Unix-оболочки, он поддерживает автодополнение имён файлов и каталогов, подстановку вывода результата команд, переменные, контроль над порядком выполнения, операторы ветвления и цикла. Ключевые слова, синтаксис и другие основные особенности языка были заимствованы из sh. Другие функции, например, история, были скопированы из csh и ksh. Bash в основном соответствует стандарту POSIX, но с рядом расширений.

Название «bash» является акронимом от англ. Bourne-again-shell («ещё-одна-командная-оболочка-Борна») и представляет собой игру слов: Bourne-shell — одна из популярных разновидностей командной оболочки для UNIX (sh), автором которой является Стивен Борн (1978), усовершенствована в 1987 году Брайаном Фоксом. Фамилия Bourne (Борн) перекликается с английским словом born, означающим «родившийся», отсюда: рождённая-вновь-командная оболочка.

В сентябре 2014 года в bash была обнаружена широко эксплуатируемая уязвимость Bashdoor.

Отличия в синтаксисе

подавляющее большинство важных скриптов командного процессора Bourne может выполняться без изменения в `bash`, за исключением тех, которые ссылаются на специальные переменные Bourne или используют встроенные команды Bourne. Синтаксис команд `Bash` включает идеи, заимствованные у Korn shell (`ksh`) и C shell (`csh`), такие как редактирование командной строки, история команд, стек каталогов, переменные `$RANDOM` и `$PPID`, синтаксис замены команды `$(...)`. Когда `Bash` используется как интерактивный командный процессор, он поддерживает автозавершение имён программ, файлов, переменных и т. п. с помощью клавиши `Tab` .

Внутренние команды

Интерпретатор `bash` имеет множество встроенных команд, часть из которых имеет аналогичные исполняемые файлы в операционной системе. Однако следует обратить внимание, что чаще всего для встроенных команд отсутствуют `man`-страницы, а при попытке просмотра справки по встроенной команде на самом деле будет выдаваться справка по исполняемому файлу. Исполняемый файл и встроенная команда могут различаться параметрами. Информация по встроенным командам расписана в справочной странице `bash`:

`man bash`

| Ввод-вывод | |
|---|--|
| <code>echo</code> | выводит выражение или содержимое переменной (<i>stdout</i>), но имеет ограничения в использовании ^[5] |
| <code>printf</code> | команда форматированного вывода, расширенный вариант команды <code>echo</code> |
| <code>read</code> | «читает» значение переменной со стандартного ввода (<i>stdin</i>), в интерактивном режиме это клавиатура |
| Файловая система | |
| <code>cd</code> | изменяет текущий каталог |
| <code>pwd</code> | выводит название текущего рабочего каталога (от <i>англ.</i> <i>print working directory</i>) |
| <code>pushd</code> | изменяет текущий каталог с возможностью возврата в обратном порядке |
| <code>popd</code> | возвращает текущий каталог после <code>pushd</code> |
| <code>dirs</code> | выводит или очищает содержимое стека каталогов, сохранённых через <code>pushd</code> |
| Действия над переменными | |
| <code>let</code> | производит арифметические операции над переменными |
| <code>eval</code> | транспирует список аргументов из списка в команды |
| <code>set</code> | изменяет значения внутренних переменных скрипта |
| <code>unset</code> | удаляет переменную |
| <code>export</code> | экспортирует переменную, делая её доступной дочерним процессам |
| <code>declare</code> , <code>typeset</code> | задают и/или накладывают ограничения на переменные |
| <code>getopts</code> | используется для разбора аргументов, передаваемых скрипту из командной строки |

| Управление сценарием | |
|--|---|
| source , . (точка) | запуск указанного сценария |
| exit | безусловное завершение работы сценария |
| exec | заменяет текущий процесс новым, запускаемым командой <code>exec</code> |
| shopt | позволяет изменять ключи (опции) оболочки "на лету" |
| Команды | |
| true | возвращает код завершения ноль (успешное завершение) |
| false | возвращает код завершения, который свидетельствует о неудаче |
| type prog | выводит полный путь к <i>prog</i> |
| hash prog | запоминает путь к <i>prog</i> |
| help COMMAND | выводит краткую справку по использованию внутренней команды <i>COMMAND</i> |
| Управление запущенными в командной оболочке задачами | |
| jobs | показывает список запущенных в командной оболочке задач либо информацию о конкретной задаче по её номеру |
| fg | переключает поток ввода на текущую задачу (или на определённую задачу, если указан её номер) и продолжает её исполнение |
| bg | продолжает исполнение текущей приостановленной задачи (или определённых задач, если указаны их номера) в фоновом режиме |
| wait | ожидает завершения указанных задач |

Скрипты

В простейшем случае, скрипт (сценарий, приказ) — простой список команд, записанный в файл. Командный процессор должен знать, что он должен этот файл обработать, а не просто прочесть его содержимое. Для этого служит специальная конструкция, называемая shebang: `#!`. Символ `#` задаёт комментарий, но в данном случае shebang означает, что после этого спецсимвола находится путь к интерпретатору для исполнения сценария.

Синтаксис

Синтаксис команд `bash` — это расширенный синтаксис команд Bourne shell. Окончательная спецификация синтаксиса команд `bash` есть в Bash Reference Manual, распространяемом проектом GNU.

```
«Hello world»
#!/usr/bin/env bash
echo 'Hello World!'
```

Этот скрипт содержит только две строки. Первая строка сообщает системе о том, какая программа используется для запуска файла. Вторая строка — это единственное действие, которое выполняется этим скриптом, он, собственно, печатает «Hello world!» в терминале.

Запуск скрипта

Для того, чтобы скрипт стал исполняемым, могут быть использованы следующие команды:

```
chmod +rx scriptname # выдача прав на чтение/исполнение любому
пользователю
chmod u+rx scriptname # выдача прав на чтение/исполнение только
"владельцу" скрипта
```

Из соображений безопасности путь к текущему каталогу `.` не включён в переменную окружения `$PATH`. Поэтому для запуска скрипта необходимо явно указывать путь к текущему каталогу, в котором находится скрипт:

```
./scriptname
```

Кроме того, передать такой файл на исполнение интерпретатору Bash можно и явно, используя команду `bash`:

```
bash scriptname
```

В этом случае не требуется ни установка прав доступа, ни использование последовательности `#!` в коде.

Перенаправление ввода-вывода

В `bash` есть встроенные файловые дескрипторы: 0 (`stdin`), 1 (`stdout`), 2 (`stderr`).

- `stdin` — стандартный ввод — то, что набирает пользователь в консоли;
- `stdout` — стандартный вывод программы;
- `stderr` — стандартный вывод ошибок.

Для операций с этими и пользовательскими дескрипторами существуют специальные символы: `>` (перенаправление вывода), `<` (перенаправление ввода). Символы `&`, `-` могут предварять номер дескриптора; например, `2>&1` — перенаправление дескриптора 2 (`stderr`) в дескриптор 1 (`stdout`).

Любая задача по написанию скрипта начинается с анализа задачи и выделении в ней более простых подзадач. Перед началом написания инструкций рекомендуется внимательно изучить задание, продумать решение каждого этапа. Работа считается выполненной, когда преподавателю продемонстрирована работа скрипта и оформлен отчет.

Отчет о лабораторной работе – технический документ, который содержит систематизированные данные о лабораторной работе, описывает теорию, используемую в лабораторной работе, ход лабораторной работы, расчеты и результаты, полученные в ходе лабораторной работы.

Отчет о лабораторной работе состоит из следующих основных элементов:

- ✓ Титульный лист
- ✓ Цель работы
- ✓ Теоретические сведения
- ✓ Расчетно-графическая часть
- ✓ Выводы по работе

Титульный лист является первой страницей отчета по лабораторной работе и служит источником информации, необходимой для поиска и обработки документа. Титульный лист обязательно должен содержать:

- ✓ Наименование вышестоящей организации
- ✓ Наименование типа учебного заведения

- ✓ Наименование учебного заведения
- ✓ Название дисциплины, по которой проводится лабораторная работа
- ✓ Номер лабораторной работы
- ✓ Название лабораторной работы
- ✓ Данные о группе и студенте (студентах), выполнивший(-их) эту работу
- ✓ Данные о преподавателе, проверяющего отчет
- ✓ Город и год

При проверке преподавателем студенческих отчетов по лабораторным работам на титульном листе преподавателем записываются замечания по отчету. Поэтому в случае необходимости переоформления отчета или внесения в содержание отчета исправлений титульный лист остается первоначальным (не заменяется новым) для того, чтобы при вторичной проверке отчета преподаватель видел все предыдущие замечания.

Цель работы указывается в точной формулировке, как указано в вашем варианте.

Теоретические сведения указываются в зависимости от поставленной задачи. В данной части требуется описать используемые команды и базовые понятия.

Расчетно-графическая часть является основной в отчете о лабораторной работе. В начале этой части указывается исходная задача, словесно описывается её анализ и выделение подзадач. Приводится решение каждой подзадачи с указанием команд и необходимых опций, используемых для этого. После этого проводится синтез решения и все приведенные ранее инструкции объединяются в единый скрипт. Приводятся снимки экрана с демонстрацией работы. Если скрипт подразумевает чтение или запись из файла, необходимо привести содержимое файлов. Если скрипт подразумевает ввод пользовательских данных или опций, то указывается с какими опциями был запущен скрипт и какие данные были введены при тестировании. Если скрипт выводит сообщение на экран, то продемонстрировать это сообщение. Для каждого скрипта должно быть проведено не менее 5 тестов с разными исходными данными и опциями.

Выводы о проделанной работе должны содержать анализ работы: выделить проблемы, с которыми пришлось столкнуться при решении задач, указать недостатки, которые имеются в вашем решении и которые требуют дополнительной доработки, какие недостатки были устранены в процессе решения задачи

Отчет оформляется в соответствии с требованиями ГОСТ Р 7.0.97-2016. Каждый раздел отчета должен иметь свой номер и заголовок, которые напечатаны жирным шрифтом 14 кеглем. Весь отчет (за исключением листинга кода) оформляется шрифтом Times New Roman. Листинг кода

оформляется шрифтом Courier New. Формулы и математические выкладки оформляются в соответствии с ГОСТ 7.32-2017.

В ходе выполнения работы приветствуется творческий подход к решению задачи. Преподавателем оценивается не только возможность программы решить поставленную задачу, но и способность программы обрабатывать ошибки и реагировать на них.

Защита работы осуществляется путем демонстрации работы ваших скриптов преподавателю на другой машине и ответами на вопросы по теме лабораторной работы.

Требования к файлам с исходным кодом и к самому коду. Ваш скрипт должен быть реализован в виде отдельного файла для каждого задания:

- .sh файлы, в которых содержится тело скрипта;

ВАЖНО! Реализация шаблонных классов должна быть в том же файле, в котором объявлен их интерфейс.

К отчету должны быть приложены файлы скриптов. Проект с исходниками может быть выгружен в репозиторий на GitHub. Имена файлов должны быть записаны только строчными латинскими буквами, для разделения можно использовать подчёркивание (`_`) или дефис (`-`). Используйте тот разделитель, который используется в проекте. Если единого подхода нет — используйте `"_"`.

Общие правила именования

1. Используйте имена, который будут понятны даже людям из другой команды.
2. Имя должно говорить о цели или применимости объекта (функции).
3. Не экономьте на длине имени, лучше более длинное и более понятное (даже новичкам) имя.
4. Меньше аббревиатур, особенно если они незнакомы вне вашего проекта.
5. Используйте только общеизвестные аббревиатуры.
6. Не сокращайте слова.

В целом, длина имени должна соответствовать размеру области видимости. Например, `n` — подходящее имя внутри функции в 5 строк, однако при описании класса это может быть коротковато.

В слове первая буква может быть заглавной (зависит от стиля: "camel case" или «Pascal case»), остальные буквы — строчные. Например, предпочтительно `ConnectTcp()`, нежелательно `ConnectTCP()`.

Имена типов начинаются с прописной буквы, каждое новое слово также начинается с прописной буквы. Подчёркивания не используются: **MyTheBestClass**, **MyTheBestStruct**.

Имена всех типов — классов, структур, псевдонимов, перечислений, параметров шаблонов — именуются в одинаковом стиле. Имена типов начинаются с прописной буквы, каждое новое слово также начинается с прописной буквы. Подчёркивания не используются.

Имена переменных (включая параметры функций) и членов данных пишутся строчными буквами с подчёркиванием между словами. Члены данных классов (не структур) дополняются подчёркиванием в конце имени. Например: **a_local_variable**, **a_struct_data_member**, **a_class_data_member_**.

Константные объекты объявляются как `constexpr` или `const`, чтобы значение не менялось в процессе выполнения. Имена констант начинаются с символа «к», далее идёт имя в смешанном стиле (прописные и строчные буквы). Подчёркивание может быть использовано в редких случаях, когда прописные буквы не могут использоваться для разделения.

Обычные функции именуются в смешанном стиле (прописные и строчные буквы); функции доступа к переменным (сеттеры и геттеры) должны иметь стиль, похожий на целевую переменную.

Общие требования ко всем скриптам:

1. Валидация аргументов:

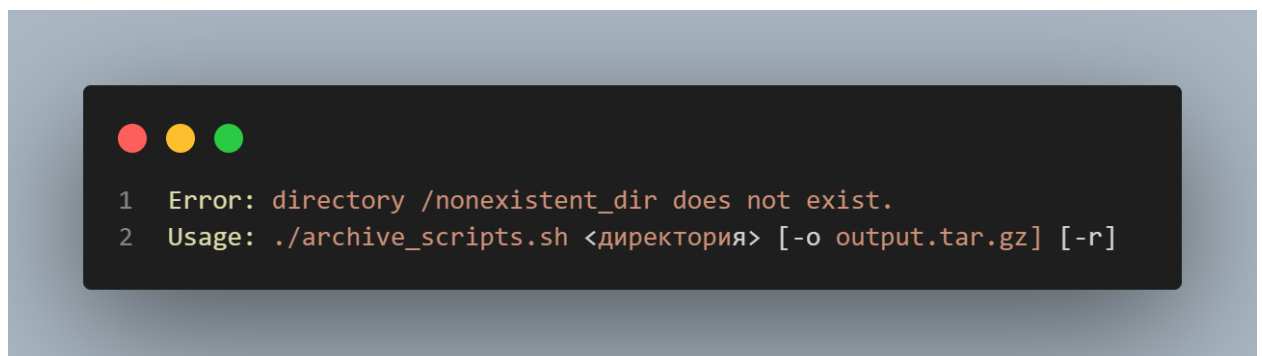
- Если аргументов недостаточно — выводить `Usage:`
`script_name <args>`.
- Проверять существование файлов/директорий.

2. Обработка ошибок:

- Если команда завершается с ошибкой — писать в `stderr` и завершаться с `exit 1`.

3. Логирование:

- Критические действия (удаление, переименование) должны логироваться.



Изображение 1 – Пример вывода ошибки

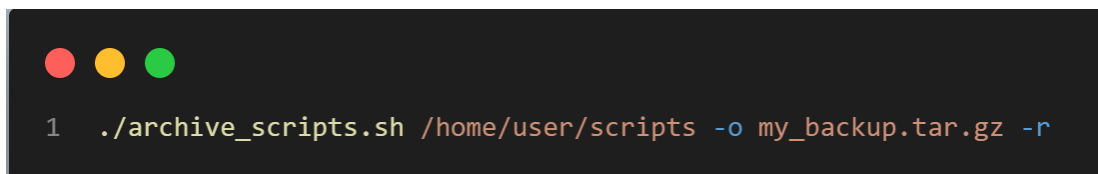
ВАРИАНТ 1. РАБОТА С ТЕКСТОВЫМИ ФАЙЛАМИ И АРХИВАМИ

1. Скрипт поиска и архивации

Написать скрипт `archive_scripts.sh`, который ищет все файлы с расширением `.sh` в указанных директориях (аргументы скрипта) и упаковывает их в архив `scripts_backup.tar.gz`.

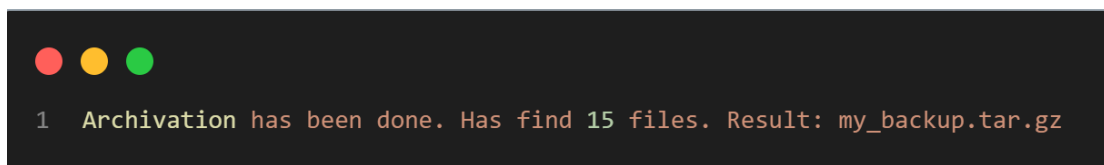
Требования:

- Скрипт принимает минимум 1 аргумент — путь к директории для поиска.
- Поддерживает опцию `-o` для указания имени выходного архива (по умолчанию `scripts_backup.tar.gz`).
- Игнорирует поддиректории, если не указана опция `-r` (рекурсивный поиск).



```
1 ./archive_scripts.sh /home/user/scripts -o my_backup.tar.gz -r
```

Изображение 2 – Пример использования



```
1 Archivation has been done. Has find 15 files. Result: my_backup.tar.gz
```

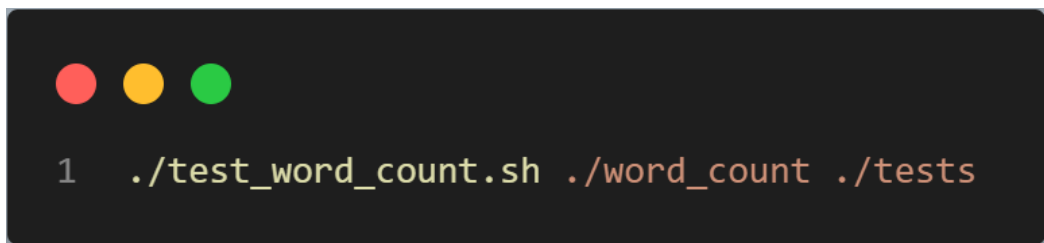
Изображение 3 – Ожидаемый вывод

2. Тестирование C++ программы.

Написать программу `word_count.cpp`, которая подсчитывает количество слов в файле. Скрипт `test_word_count.sh` должен генерировать 5 тестовых файлов, запускать программу и сравнивать её вывод с ожидаемым результатом.

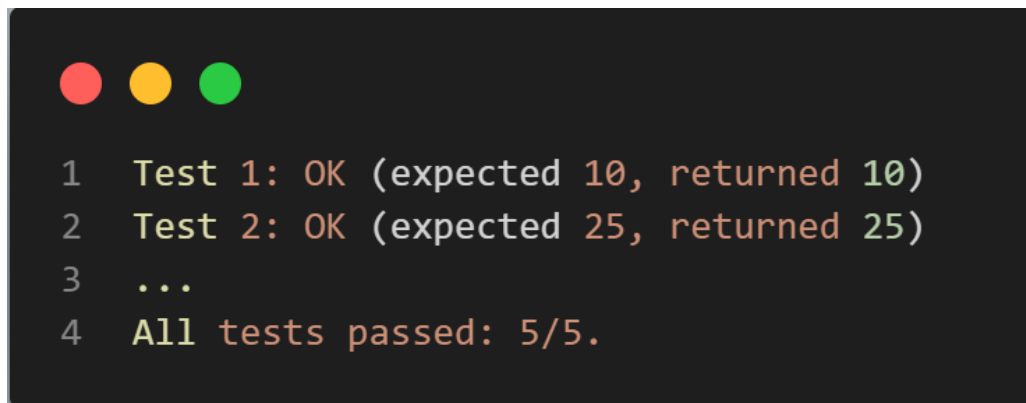
Требования:

- `word_count.cpp` принимает 1 аргумент — путь к файлу, возвращает число слов.
- `test_word_count.sh` принимает 2 аргумента:
 - Путь к исполняемому файлу `word_count`.
 - Директорию для сохранения тестов (если не существует — создает).



```
1 ./test_word_count.sh ./word_count ./tests
```

Изображение 4 – Пример использования



```
1 Test 1: OK (expected 10, returned 10)
2 Test 2: OK (expected 25, returned 25)
3 ...
4 All tests passed: 5/5.
```

Изображение 5 – Ожидаемый вывод

3. Обработка логов

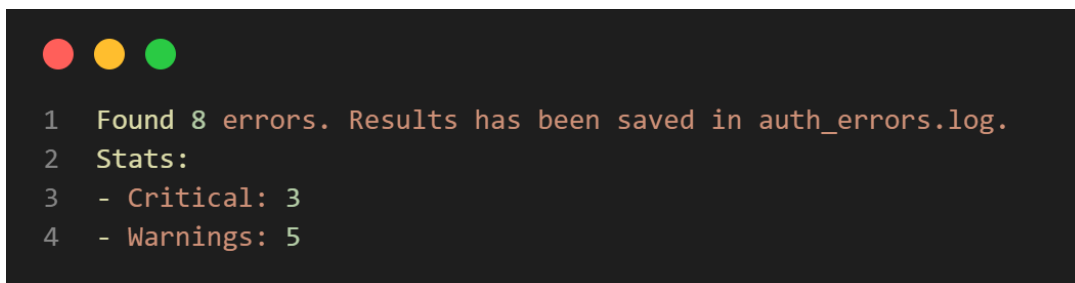
Написать скрипт `log_analyzer.sh`, который анализирует файл `/var/log/syslog` (или любой другой), фильтрует строки с ошибками (`grep -i "error"`), сохраняет их в отдельный файл и выводит статистику (например, количество ошибок).

Требования:

- Принимает 1 аргумент — путь к лог-файлу (по умолчанию `/var/log/syslog`).
- Поддерживает опцию `-o` для указания выходного файла (по умолчанию `errors.log`).

```
./log_analyzer.sh /var/log/auth.log -o auth_errors.log
```

Изображение 6 – Пример использования



```
1 Found 8 errors. Results has been saved in auth_errors.log.
2 Stats:
3 - Critical: 3
4 - Warnings: 5
```

Изображение 7 – Ожидаемый вывод

ВАРИАНТ 2. УПРАВЛЕНИЕ ПРОЦЕССАМИ И ПРАВАМИ

1. Скрипт мониторинга процессов

Написать скрипт `process_monitor.sh`, который принимает имя процесса и выводит его PID, потребление CPU и памяти.

Требования:

- Принимает 1 аргумент — имя процесса (например, `nginx`).
- Если процесс не указан — выводит список всех процессов

Пример использования:

```
./process_monitor.sh nginx
```

Ожидаемый вывод (пример):

```
Process: nginx
```

```
PID: 1234
```

```
CPU: 2.5%
```

```
Memory: 120 MB
```

2. Тестирование сортировки.

Написать программу `sort_numbers.cpp`, которая сортирует числа из файла с применением сортировки слиянием. Скрипт `test_sort.sh` должен генерировать случайные числа, запускать программу и проверять корректность сортировки.

Требования:

- `sort_numbers.cpp` принимает 1 аргумент — входной файл с числами.
- `test_sort.sh` принимает 1 аргумент — количество тестов (по умолчанию 5).

Пример использования:

```
./test_sort.sh 10
```

Ожидаемый вывод (пример):

```
Test 1: OK
```

```
...
```

```
Test 10: OK
```

```
PASSED: 10/10
```

3. Изменение прав доступа

Написать скрипт `fix_permissions.sh`, который рекурсивно меняет права доступа для всех файлов в директории (например, 644 для файлов, 755 для директорий).

Требования:

- Принимает 1 аргумент — путь к директории.
- Поддерживает опцию -v (verbose-режим, вывод изменений).

Пример использования:

```
./fix_permissions.sh /home/user/project -v
```

Ожидаемый вывод (пример):

```
Changes at: /home/user/project/file1.txt (644)
```

```
Changes at: /home/user/project/dir (755)
```

```
...
```

```
DONE! Overall changed 23 directories and 186 files
```

ВАРИАНТ 3. ПОИСК И ОБРАБОТКА ФАЙЛОВ

1. Поиск дубликатов

Написать скрипт `find_duplicates.sh`, который ищет файлы с одинаковым содержимым (используя `md5sum`).

Требования:

- Принимает 1 аргумент — директорию для поиска.
- Выводит список файлов с одинаковым содержимым.

Пример использования:

```
./find_duplicates.sh /home/user/documents
```

Ожидаемый вывод (пример):

```
Found duplicates:
```

```
- /home/user/documents/file1.txt  
- /home/user/documents/backup/file1_copy.txt
```

2. Тестирование поиска подстроки.

Написать программу `substring_search.cpp`, которая ищет подстроку в файле. Скрипт `test_search.sh` должен создавать тестовые файлы и проверять работу программы.

Требования:

- `substring_search.cpp` принимает 2 аргумента: файл и подстроку.
- `test_search.sh` генерирует тесты автоматически.

Пример использования:

```
./test_search.sh
```

Ожидаемый вывод (пример):

```
Test "search 'hello' in test1.txt": OK (find 3 entries)
```

3. Переименование файлов

Написать скрипт `batch_rename.sh`, который переименовывает все файлы в директории, добавляя префикс/суффикс (например, `file_1.txt` → `backup_file_1.txt`).

Требования:

- Принимает 3 аргумента:
 - Директория.
 - Префикс (например, `backup_`).
 - Суффикс (например, `_2023`).

Пример использования:

```
./batch_rename.sh      /home/user/photos      "vacation_"  
"_summer"
```

Ожидаемый вывод (пример):

File photo1.jpg → vacation_photo1_summer.jpg

File photo2.jpg → vacation_photo2_summer.jpg

ВАРИАНТ 4. РАБОТА С CSV И AWK

1. Скрипт обработки CSV

Написать скрипт `csv_stats.sh`, который читает CSV-файл и выводит статистику (например, среднее значение в столбце).

Требования:

- Принимает 2 обязательных аргумента:
 - Путь к CSV-файлу.
 - Номер столбца для анализа (начиная с 1).
- Опционально `-d` (разделитель, по умолчанию `“;”`).

Пример использования:

```
./csv_stats.sh data.csv 3 -d ";"
```

Ожидаемый вывод (пример):

```
Stats for column 3:
```

```
Average: 42.5
```

```
Maximum: 100
```

```
Minimum: 10
```

2. Тестирование парсера CSV.

Написать программу `csv_parser.cpp`, которая извлекает данные из CSV. Скрипт `test_csv.sh` должен проверять корректность работы.

Требования:

- `csv_parser.cpp` принимает 3 аргумента:
 - Входной CSV-файл.
 - Номер столбца.
 - Разделитель.
- `test_csv.sh` принимает 1 аргумент — путь к бинарнику `csv_parser`

Пример использования:

```
./test_csv.sh ./csv_parser
```

Ожидаемый вывод (пример):

```
Test 1: OK (col 2, value=42)
```

```
Тест 2: OK (col 3, value="text")
```

```
PASSED: 2/2 Tests
```

3. Генерация отчета

Написать скрипт `report_generator.sh`, который формирует отчет о файлах в директории (размер, дата изменения) и сохраняет его в формате Markdown.

Требования:

- Принимает 1 аргумент — директорию для анализа.
- Опция `--format` (формат вывода: `txt` или `md`, по умолчанию `md`).

Пример использования:

```
./report_generator.sh ~/project --format txt
```

Ожидаемый вывод (пример):

```
Report for /home/user/project:
```

```
Files: 15
```

```
Total size: 2.4 MB
```

ВАРИАНТ 5. АВТОМАТИЗАЦИЯ СБОРКИ И ТЕСТИРОВАНИЯ

1. Скрипт сборки проекта

Написать скрипт `build_project.sh`, который компилирует все `.cpp` файлы в директории.

Требования:

- Принимает 1 аргумент — директорию с исходниками.
- Опции:
 - `-o` (имя выходного файла, по умолчанию `a.out`).
 - `-c` (флаг очистки, удаляет `.o` файлы).

Пример использования:

```
./build_project.sh src/ -o app -c
```

Ожидаемый вывод (пример):

```
Compilation is done. Executable file: app
Deleted 10 .o file(s).
```

2. Тестирование калькулятора.

Написать программу `calculator.cpp` (сложение, вычитание). Скрипт `test_calc.sh` должен генерировать тесты и проверять ответы.

Требования:

- `calculator.cpp` принимает 3 аргумента:
 - Число A.
 - Оператор (+, -, *, /).
 - Число B.
- `test_calc.sh` принимает 0 аргументов, генерирует и запускает тесты автоматически (проводит не менее 100 тестов).

Пример использования:

```
./test_calc.sh
```

Ожидаемый вывод (пример):

```
Test 1: 2 + 2 = 4 [OK]
Test 2: 5 * 3 = 15 [OK]
...
PASSED: 100/100
```

3. Управление сервисами

Написать скрипт `service_control.sh`, который проверяет статус сервиса (например, `nginx`) и перезапускает его, если он не работает

Требования:

- Принимает 1 аргумент — имя сервиса (например, nginx).
- Опции:
 - -s (действие: start, stop, restart).
 - -v (подробный вывод).

Пример использования:

```
./service_control.sh nginx -s restart -v
```

Ожидаемый вывод (пример):

```
Service nginx restarted (PID: 1234).  
Status: active (running).
```

ВАРИАНТ 6. ПОИСК И ЗАМЕНА ТЕКСТА В ФАЙЛАХ

1. Скрипт для замены текста

Написать скрипт `replace_text.sh`, который во всех файлах в директории заменяет искомую подстроку на заданную.

Требования:

- Принимает 3 аргумента:
 - строка_поиска
 - строка_замены
 - файл (или директория, если указан `-r`)
- Опция `-r` — рекурсивный поиск в поддиректориях.

Пример использования:

```
./replace_text.sh "old" "new" file.txt -r
```

Ожидаемый вывод (пример):

```
Replaced 5 entries в file.txt
```

2. Тестирование программы замены текста.

Написать программу `text_replacer.cpp`, которая в заданном файле находит все вхождения искомой подстроки и заменяет их на заданную. Скрипт `test_replacer.sh` должен генерировать тесты и проверять ответы.

Требования:

- `text_replacer.cpp` принимает 3 аргумента:
 - входной файл,
 - строку поиска,
 - строку замены.
- `test_replacer.sh` создает тестовые файлы и проверяет работу программы.

Пример использования:

```
./test_replacer.sh
```

Ожидаемый вывод (пример):

```
Test 1: OK
```

```
Test 2: OK
```

```
...
```

```
PASSED: 10/10
```

3. Подсчет строк в файлах

Написать скрипт `service_control.sh`, который проверяет статус сервиса (например, `nginx`) и перезапускает его, если он не работает

Требования:

- Принимает 1 аргумент — файл или директорию
- -r для директории – подсчитать строки в каждом файле, в том числе во всех вложенных каталогах.
- --detail – выводит информацию о каждом файле отдельно
- Выводит общее количество строк.

Пример использования:

```
./line_counter.sh /home/user/docs -r
```

Ожидаемый вывод (пример):

```
Found 42 lines in 3 file(s).
```

ВАРИАНТ 7. РАБОТА С АРХИВАМИ

1. Скрипт распаковки

Написать скрипт `unpack.sh`, который извлекает содержимое в целевую директорию.

Требования:

- Принимает 2 аргумента
 - файл архива (tar.gz, zip).
 - целевая директория (если не указана, то в текущую папку)

Пример использования:

```
./unpack_archives.sh backup.tar.gz
```

Ожидаемый вывод (пример):

```
Unpacked 10 files from backup.tar.gz.
```

2. Тестирование программы подсчёта веса Хемминга.

Написать программу `hamming_weight.cpp`, которая вычисляет вес Хэмминга (сумма всех единичных битов) для заданного числа. Скрипт `test_hamming.sh` долже проверять ответы.

Требования:

- `test_hamming.sh` принимает 2 аргумента:
 - файл программы
 - каталог с тестами (файл теста содержит число и его вес Хэмминга)

Пример использования:

```
./test_hamming.sh ./hamming_weight ./tests
```

Ожидаемый вывод (пример):

```
Test 1: OK
```

```
Test 2: OK
```

```
...
```

```
PASSED: 10/10
```

3. Проверка целостности архива

Написать скрипт `check_archive.sh`, который проверяет целостность архивов в каталоге

Требования:

- Принимает 1 аргумент — путь к каталогу.
- Имеет опцию `-r` – проверять все архивы во вложенных каталогах
- Проверяет, что архив не поврежден..

Пример использования:

```
./check_archive.sh ./data -r
```

Ожидаемый вывод (пример):

```
Archive ./data/backup.zip Status: OK
```

```
Archive ./data/old/backup_23.02.2016.zip Status:  
DAMAGED
```

```
...
```

```
4 archive(s) has been checked
```

```
3 archive(s) OK
```

```
1 archive(s) DAMAGED
```

ВАРИАНТ 8. МАРКИРОВКА ДАННЫХ

1. Скрипт добавления timestamp

Написать скрипт `add_timestamp.sh`, который добавляет в конец каждого текстового файла текущий временной штамп.

Требования:

- Принимает хотя бы один аргумент
 - Имена текстовых файлов передаются скрипту в качестве параметров, их число заранее не известно.
- Принимает опцию `-d` после которой указывается директория, в которой осуществляется поиск всех текстовых файлов

Пример использования:

```
./add_timestamp.sh ./my_code.cpp ./data/my_file.txt
```

Ожидаемый вывод (пример):

```
Current timestamp 10/02/2024 has been added in 10 files
```

2. Тестирование программы сортировки.

Написать программу `insertion_sort.cpp`, считывает данные и сортирует их в порядке возрастания, используя сортировку вставками. Скрипт `test_sort.sh` должен проверять ответы.

Требования:

- `test_sort.sh` принимает 2 аргумента:
 - файл программы
 - каталог с тестами (файл теста содержит набор неупорядоченных чисел)

Пример использования:

```
./test_sort.sh ./insertion_sort ./tests
```

Ожидаемый вывод (пример):

```
Test 1: OK
```

```
Test 2: OK
```

```
...
```

```
PASSED: 10/10
```

3. Маркировка файлов по содержимому

Написать скрипт `add_copyright.sh`, который добавляет в конец каждого файла, содержащего заданную подстроку, текущий временной штамп и имя пользователя.

Требования:

- Принимает хотя 2 аргумента

- искомая подстрока.
- Каталог, в котором необходимо искать файлы
- опция -r (поиск рекурсивно во всех подкаталогах)
- опция --detail (выводит список всех файлах, в который были внесены изменения)

Пример использования:

```
./add_copyright.sh ./source -r
```

Ожидаемый вывод (пример):

```
Total 14 files has been marked
```

ВАРИАНТ 9. УПРАВЛЕНИЕ ФАЙЛАМИ

1. Скрипт для копирования с фильтром

Написать скрипт `filtered_copy.sh`, который копирует файлы из каталога в другой каталог.

Требования:

- Принимает два аргумента
 - Исходная директория
 - Целевая директория
- Принимает опцию `-e` после которой указывается расширение файлов, которые должны быть скопированы

Пример использования:

```
./filtered_copy.sh ./my_folder ./new_my_folder -e cpp
```

Ожидаемый вывод (пример):

```
12 files has been copied!
```

2. Тестирование программы сортировки.

Написать программу `quicksort.cpp`, считывает данные и сортирует их в порядке возрастания, используя быструю сортировку. Скрипт `test_sort.sh` должен проверять ответы.

Требования:

- `test_sort.sh` принимает 2 аргумента:
 - файл программы
 - каталог с тестами (файл теста содержит набор неупорядоченных чисел)

Пример использования:

```
./test_sort.sh ./quicksort ./tests
```

Ожидаемый вывод (пример):

```
Test 1: OK
```

```
Test 2: OK
```

```
...
```

```
PASSED: 10/10
```

3. Удаление пустых файлов

Написать скрипт `clean_empty.sh`, который удаляет пустые файлы

Требования:

- Принимает 1 аргумент
 - Целевая директория.
- опция `-r` (поиск рекурсивно во всех подкаталогах)

- опция `--detail` (выводит список всех файлов, в которые были удалены)

Пример использования:

```
./clean_empty.sh ./source -r
```

Ожидаемый вывод (пример):

```
Total 14 empty files has been deleted
```

ВАРИАНТ 10. УПРАВЛЕНИЕ ПРОЦЕССАМИ

1. Скрипт для копирования с фильтром

Написать скрипт `greed_processes.sh`, который выведет `top N` процессов, которые потребляют больше всего памяти и процессора в системе в виде таблицы

Требования:

- Принимает один аргумент
 - Количество процессов `N`
- Принимает опцию `-mem`, после которой выводится `top N` процессов, которые потребляют больше всего памяти.
- Принимает опцию `-cpu`, после которой выводится `top N` процессов, которые потребляют больше всего процессора.
- Использование двух опций вместе равносильно их отсутствию: выводятся две таблицы: по памяти и по загрузке процессора.

Пример использования:

```
./greed_processes.sh 5 -mem
```

Ожидаемый вывод (пример):

Top-5 processes by memory usage:

| ----- | | | |
|-------|------|----------|-------------|
| PID | %MEM | RSS (KB) | COMMAND |
| 1234 | 5.2 | 1024 | chrome |
| 5678 | 4.8 | 980 | java |
| 9012 | 3.5 | 720 | mysqld |
| 3456 | 2.1 | 420 | gnome-shell |
| 7890 | 1.8 | 360 | slack |

2. Тестирование программы сортировки.

Написать программу `shellsort.cpp`, считывает данные и сортирует их в порядке возрастания, используя сортировку Шелла. Скрипт `test_sort.sh` должен проверять ответы.

Требования:

- `test_sort.sh` принимает 2 аргумента:
 - файл программы
 - каталог с тестами (файл теста содержит набор неупорядоченных чисел)

Пример использования:

```
./test_sort.sh ./shellsort ./tests
```

Ожидаемый вывод (пример):

Test 1: OK

Test 2: OK

...

PASSED: 10/10

3. Удаление пустых файлов

Написать скрипт `process_tree.sh`, который выводит дерево процесса по заданному номеру или имени процесса. То есть отображает родителя процесса и всех своих потомков и их потомков и так далее.

Требования:

- Принимает 1 аргумент
 - Имя процесса ИЛИ его PID.
- опция `--detail` (выводит дополнительную информацию о процессах)

Пример использования:

```
./process_tree.sh 1234
```

Ожидаемый вывод (пример):

Process tree for nginx (PID: 1234):

Parent process: 1 (systemd)

```
├─ 1234 (nginx)
  │
  ├─ 1235 (nginx)
  │
  ├─ 1236 (nginx)
  │
  └─ 1237 (nginx)
```