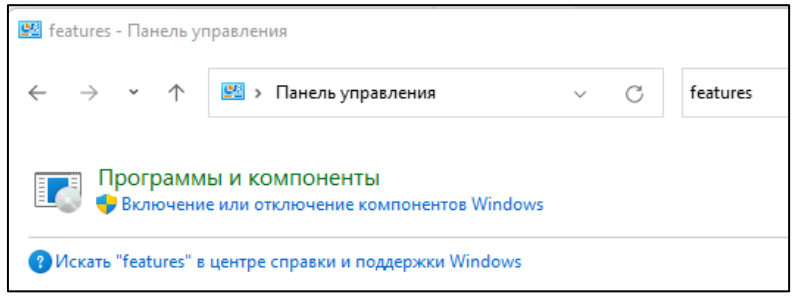
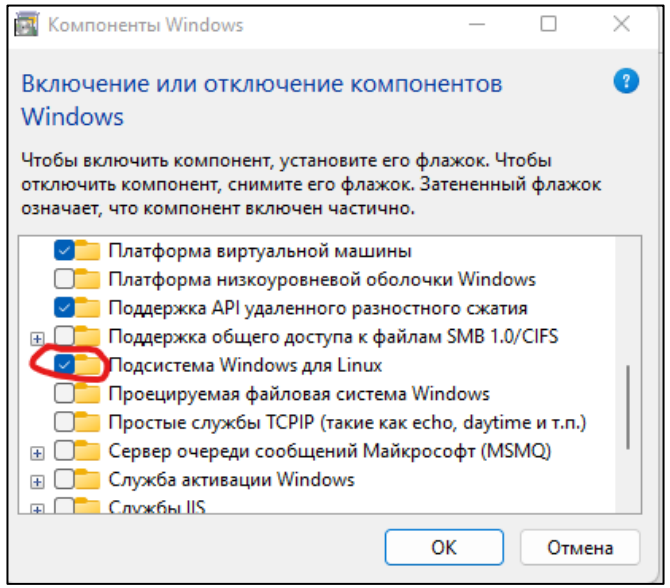
UNIX SHELL SCRIPTS

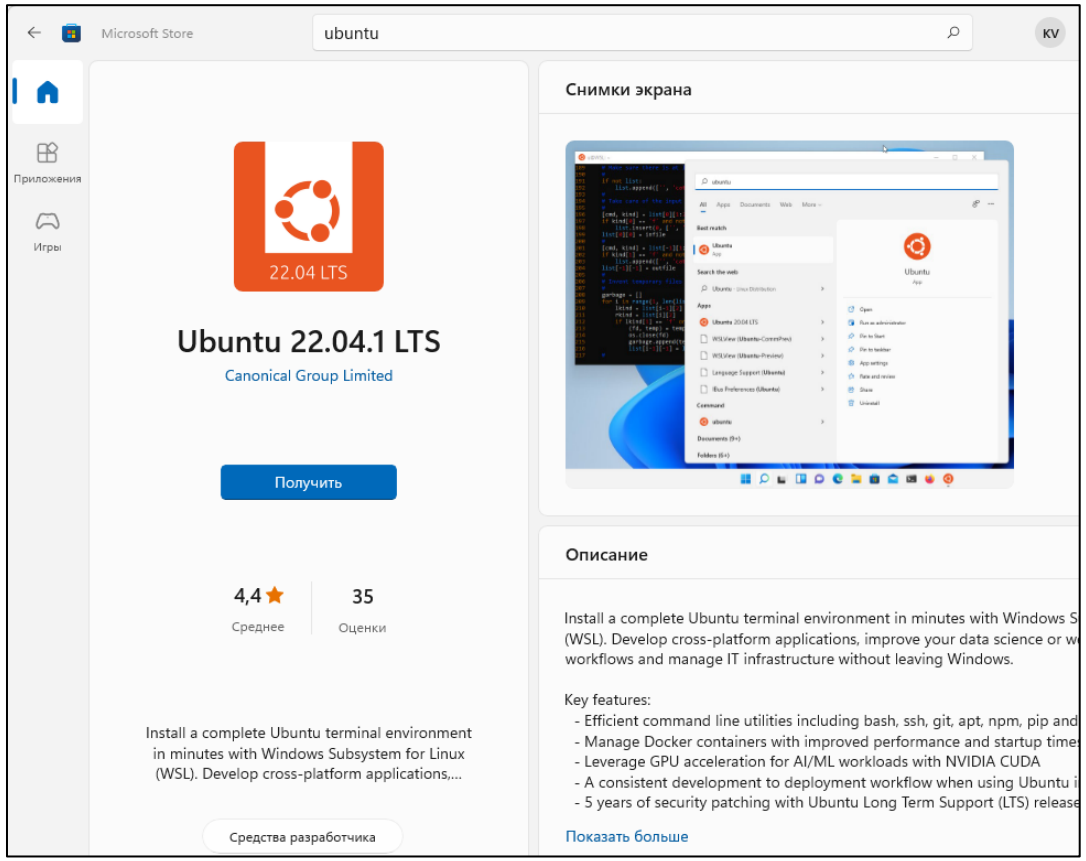
Эффективная работа с консолью

К. Владимиров, Syntacore, 2022
mail-to: konstantin.vladimirov@gmail.com

Как начать под Windows

1. 

2. 

3. 

Менеджер пакетов apt (Ubuntu)

- Работа с пакетами требует привилегий sudo:

```
$ sudo apt update
$ sudo apt upgrade
```

- Если вам нужна конкретная программа, её установка из пакетного менеджера очень проста.

```
$ sudo apt install gcc
```

- Вы можете также искать программу в репозитории по регулярке.

```
$ apt search --names-only '^gcc-?[0-9]*$'
```

Для этой лекции

- Поставьте следующие программы:

```
gcc (компилятор)
vim (редактор)
```

Shell invitation

- Вы внутри оболочки:

```
$ команда
```

Обсуждение

- Что выводит эта программа?

```
$ cat hellor.c

#include <stdio.h>

int main() {
  printf("Hello, world!\r");
}

$ gcc hellor.c -o hellor

$ ./hellor
```

что будет здесь?

Новые команды на этом слайде

- Вывод файла на stdout.

```
cat <file>
```

- Компиляция и линковка программы на языке C.

```
gcc <source> [-o <executable>]
```

- Запуск относительно текущей папки.

```
./<executable>
```

Фокус с исчезновением

- Продолжаем эксперименты.

```
$ ./hellor
$ ./hellor > hello.out
$ vim hello.out
```

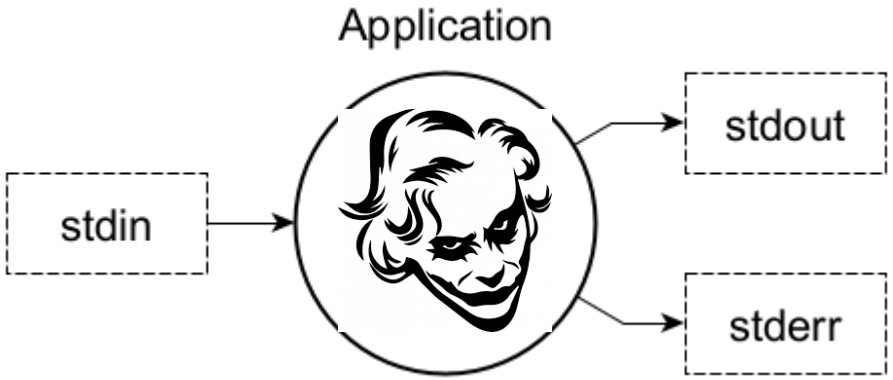
- Перенаправление stdout.

```
<executable> > <file>
```

- Консольный редактор.

```
vim <file>
```

- Вывод на stdout и stderr.



- Это потоки вывода. Но куда они направлены?
- Внезапно – в любой файл.

Идеология Unix: всё есть файл

```
/
  /dev/
  /etc/
  /home/
    /home/tilir
  /mnt/
    /mnt/c/
  /usr/
    /usr/bin/
    /usr/lib/
    /use/include
```

Навигация по дереву файлов

```
$ pwd (показать где я сейчас)
/mnt/c/research/c-graduate/bash
$ ls (показать файлы в текущей папке)
README.md  hello.out  hellor  hellor.c
$ cd /mnt/c/research/c-graduate
```

- Также есть две специальных папки:

```
$ cd . (остаться в текущей папке)
$ cd .. (перейти на уровень выше)
```

Аргументы команд

- Настоящий синтаксис команды ls

```
ls [OPTION]... [FILE]...
```

- Выяснение всех опций

```
$ man ls
```

- Например попробуйте:

```
$ ls -l .
$ ls -la ../*
$ ls -d ../*
```

Аргументы компилятора

```
gcc <source> [OPTIONS]
```

- Популярные опции

```
-o, -g, -O0, -O2, -Wall, -lm
```

- Без оптимизаций, с libm и отладочной информацией.

```
gcc -O0 -g my.c -o my.x -lm
```

- С оптимизациями и всеми warnings.

```
gcc -O2 -Wall my.c -o my.x -lm
```

Сразу три потока ввода-вывода

```
int main() {
  int n, res;
  res = fscanf(stdin, "%d", &n);

  if (res != 1) {
    fprintf(stderr, "Error: input incorrect\n");
    abort();
  } else if (n == 0) {
    fprintf(stderr, "Error: division by zero\n");
    abort();
  }

  fprintf(stdout, "%d\n", 720 / n); // просто printf
}
```

Перенаправления

```
$ ./allthree < 001.in 1> 001.out 2> 001.err
```

- Вместо того, чтобы набивать руками с клавиатуры, мы готовим файл и потом перенаправляем стандартный ввод и поток ошибок.

```
$ ./allthree < 001.in 1> 001.log 2>1
```

- Тут важен порядок: вы должны сначала перенаправить поток в файл, потом направить в него другой поток.
- Можно перенаправить оба сразу.

```
./allthree < 001.in >& 001.log
```

Ваши сообщения и shell сообщения.

- Shell тоже может выводить информацию

```
$ ./allthree < 002.in
Error: division by zero
Aborted

$ echo $?
134
```

- Оболочка вмешивается

```
$ ./allthree < 002.in >& all.log
Aborted
```

- Как починить вещи?

Новые команды

`echo argument`

- Выводит аргумент на stdout.

```
$ echo "Hello, world!"
```

`$?` код завершения последней команды.

```
int main() { return 10; }

$ ./ret10

$ echo $?
```

Вложенные оболочки

- Специальный синтаксис для вложенной оболочки.

```
$ ( ./allthree < 002.in ) >& all.log
Aborted
```

- Не помогает т.к. вложенная оболочка возвращает код последней операции.
- Но мы можем изменить код чего угодно на `0` или `1` логикой.

```
$ ( ./allthree < 002.in || false ) >& all.log
```

```
$ echo &? # догадайтесь что выведет
```

- Теперь shell message остаётся во внутренней оболочке

Автоматизация тестирования

```
$ ./allthree < 001.in > 001.outerr 2>1
```

- Допустим мы приготовили много файлов:

```
$ ls tests
001.in 002.in 003.in
$ ./allthree < tests/001.in
```

- Хотелось бы прогнать их все вместе.

Переменные и циклы в консоли

- Рассмотрим следующую строчку.

```
$ for i in tests/*; do echo $i; done
```

- Новая концепция: переменная `i`. Выражение `$i` это её значение.

```
$ myvar=10
$ echo $myvar # более точно: ${myvar} (сравним: ${my}var)
10
```

- Цикл for-in итерирует переменную по множеству значений. Сделаем его тело более интересным.

```
$ for i in tests/*; do echo $i; ./allthree < $i; done
```

Точка с запятой в bash

- Точка с запятой может отражать последование.

```
$ pwd; whoami
```

- Но может быть специальным синтаксическим элементом.

```
for i in tests/*; do echo $i; done
```

- Внезапно в этих двух случаях есть не только различие, но и сходство.
- Догадаетесь?

Пользователи

```
$ whoami
```

- Выводит текущего пользователя.
- Постоянная переменная окружения `$HOME`
- Содержит путь до "домашней" папки текущего пользователя.

`cd ~` или `cd $HOME`

Концепция PATH

- Почти любая команда это исполняемый файл который где-то лежит.

```
$ which gcc

/usr/bin/gcc
```

- Оболочка догадывается где он, так как путь `/usr/bin` добавлен в PATH.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
```

- Поэтому вы запускаете gcc просто, а ваши команды с относительным путём.

```
$ gcc hello.c -o hello.x # считай из PATH откуда брать gcc
$ ./hello.x              # не читай PATH, я сам скажу
```

Обсуждение: простая задача

- Вам нужно создать в папке `$folder` файлы `1.in`, `2.in`, `3.in` и т. д. до `$border`. В каждый из них должно быть записано значение: 1, 2, 3 и т. д.
- Давайте выучим новую команду seq которая строит последовательность.

```
$ border=3

$ seq 1 1 $border (то же что for i in 1 2 3; do echo $i; done)
1
2
3
```

- Как вы напишете bash for loop?

Проблемы с однострочниками

- Однострочный скрипт решающий задачу с прошлого слайда.

```
$ mkdir -p $folder && for i in $(seq 1 1 $border);\
> do echo "$i.in"; echo $i > "myfolder/$i.in"; done
```

- В нём есть своя суровая красота. Но есть и проблемы.

Новые команды

`&&` выполнить следующую команду только если выполнена предыдущая.

`$(some command)` использовать выдачу stdout как переменную.

`mkdir -p` создать папку, если её ещё нет.

- Альтернативой написанию однострочников служит написание скриптов.

Первый скрипт, первые неудачи

```
$ cat first.sh

echo "folder is $folder"
echo "border is $border"

$ folder=myfolder
$ border=10

$ bash first.sh
folder is
border is
```

Идея unix shell

- Есть разные shells: sh, zsh, tcsh, ….
- На этом занятии мы говорим о bash.
- Когда в Ubuntu вы находитесь "в консоли" реально вы в оболочке (по умолчанию в bash).

```
bash [OPTIONS] COMMAND
```

- Запускает вложенную оболочку и исполняет в ней команду.

Export и source

```
$ folder=myfolder
$ border=10

$ source first.sh
folder is myfolder
border is 10

$ export folder=myfolder
$ export border=10

$ bash first.sh
folder is myfolder
border is 10
```

Новые команды

```
source file [arguments]
```

- Исполняет команды из файла в текущей оболочке.

```
export [name[=value]...]
```

- Маркирует каждое имя как экспортное, то есть видимое порождённым процессам.

```
$ export -p # весь текущий экспорт
```

Скрипт для простой задачи

- Теперь мы можем написать скрипт.

```
$ cat second.sh

#!/bin/bash
# shebang указывает исполнитель

mkdir -p $folder

for i in $(seq 1 1 $border)
do
  echo "processing: $i"
  echo $i > "myfolder/$i.in"
done
```

Атрибуты файлов

```
$ ls -la second.sh
-rwxrwxrwx .... second.sh

owner group other
 rwx   rwx   rwx
```

- Например что означает?
  `-rwxr--r-- .... second.sh`
- Сделать файл исполняемым.

```
$ chmod "+x" myfile
```

Пример: скрипт на руби и пайтоне

- Поскольку мы будем его запускать через bash, шебанг указывает руби или пайтон.

```
$ cat rubyexample.rb

#!/usr/bin/ruby
puts "Hello, world!"

$ cat pyexample.py

#!/usr/bin/env python3
print("Hello World")
```

- Во многих скриптовых языках стиль комментов копирует bash ради шебанга.

Запуск с env

- Слева для руби шебанг хуже чем для python.
- Дело в том что шебанг умеет только абсолютные пути.

```
$ /usr/bin/env python3
```

- Запустит python используя $PATH
- Теперь неважно где стоит python, он будет найден.

Параметры скрипта

- Зависимость от переменных окружения таких как myfolder это не особо ок.

```
$ env folder=myfolder border=10 ./first.sh

$ cat third.sh

#!/bin/bash

echo "param 0 is: $0"
echo "param 1 is: $1"

$ ./third.sh myfolder

param 0 is: ./third.sh
param 1 is: myfolder
```

Специальные переменные

- Параметры передаются через

`$0` (название скрипта)

`$1` (первый аргумент)

- И так далее. Но это не очень удобно для обычных случаев.

Позиционные аргументы

```
$ cat positional.sh

#!/bin/bash

while getopts b:f: flag
do
  case ${flag} in
    b) border=${OPTARG};;
    f) folder=${OPTARG};;
  esac
done

echo "folder is $folder"
echo "border is $border"
```

Новые команды

`getopts opts name [arg]`

- Разбирает позиционные параметры как опции.

`case (оператор, не команда)`

- Выполнение действий (разделитель `;;`) в зависимости от аргумента.
- Плюс ещё один вид циклов: `while`

Ещё немного циклов

- В мире unix shell любят всякие циклы.

```
i=1;
j=$#;

while [ $i -le $j ]
do
  echo "Param $i: $1";
  i=$((i + 1));
  shift 1;
done
```

Новые команды

`$#` количество аргументов

`[ expr ]` то же что `test "expr"`

`$((expr))` арифметическая подстановка

- Например: `for ((i=0; i<10; i++))`

`shift N` сдвиг аргументов на N позиций

```
$ ./shifted.sh a b "c d" 'e f' g "h" i 'j' # что на экране?
```

Слабая динамическая типизация

- Переменные bash слабо типизированы: то как они трактуются зависит от их применения.

```
$ border=10
$ echo "a${border}b"
a10b
$ echo $(( 5 + $border ))
15
```

- Каждая переменная может быть переопределена с другим типом.

```
$ if [ "x$border" = "x" ]; then echo "Yes"; fi
$ unset border
$ if [ "x$border" = "x" ]; then echo "Yes"; fi # Yes
```

Разница между кавычками

- На прошлом слайде мы использовали кавычки взаимозаменяемо.

```
$ border=10

$ echo "$border"
10

$ echo '$border'
$border
```

- Разница в раскрытии переменных и трактовке спецсимволов.

Обсуждение: суперoружие

- Есть несколько типичных задач, писать скрипты для которых на bash было бы долго и сложно.
- Но поскольку unix shell развивается с 70-х, все эти задачи давно известны. И для каждого такого класса известна утилита.
- Очень часто это даёт нам в руки действительно большую пушку!

Супероружие: grep

- Несмотря на название (global regular expression print) чаще всего греп используется для поиска.

```
$ grep rec *

fibdbl.c:// Double precision fibonacci. Try n = 90

fibnaive.c:// Naive recursive fibonacci. Try n = 50.
```

- Слово "грепнуть" это такая же идиома, как "загуглить".
- Самые частые опции это `-r`, `-i`, `-I`.
- Также полезны -A и -B чтобы захватывать куски до и после.

Супероружие: sed

- Здесь название (stream editor) не подводит: мы в основном делаем в файлах массированные замены.
- Например вы хотите переименовать в файле myvar в theirvar.

```
sed -i 's/myvar/theirvar/g' myfolder/1.in
```

- Здесь -i означает менять файл сразу на месте.
- Спецификация `s/pattern/replace/where` означает заменить pattern на replace.

Супероружие: awk

- AWK это специальный язык обработки данных.

```
$ awk '{print "Hello, world!"}'
```

- Очень часто его используют как парсер по столбцам:

```
$ ls -la | awk '{ print $9}'
```

- Разделители можно программировать.

```
$ echo "My:name:is:Tom" | awk -F: '{$4="Adam"; print $0}'
```

- Можно задавать "шапку"

```
$ echo "A B C D" | awk 'BEGIN {print "--"} {$4="E"; print $0}'
```

Комбинации команд

- Одна команда может обрабатывать stdout другой через pipe.

```
$ find -name "*.c" | grep -i sort
./03-arrays/inssort.c
./03-arrays/qsort.c
./03-arrays/selsort.c
./toolchain/gdb/buggy-sort.c
./toolchain/gdb/sort-smash.c
```

- Это позволяет не засорять оболочку переменными.
- Но теперь проблема как получить только пути и оставить только уникальные?

```
dirname [OPTION] NAME... # не принимает имя со stdin
```

Волшебство xargs

```
# список папок в которых есть файлы с sort в названии
$ find -name "*.c" | grep -i sort | xargs dirname | sort | uniq
./03-arrays
./toolchain/gdb
```

- Полный синтаксис `xargs [OPTION]... COMMAND [INITIAL-ARGS]...`
- Вызывает команду и делает из stdin её аргументы.

Массивы в bash

```
logPaths=("api.log" "auth.log" "jenkins.log" "data.log")
logEmails=("anna@email" "bob@email" "jon@email" "emma@email")

for i in ${!logPaths[@]}; do
  log=${logPaths[$i]}
  stakeholder=${logEmails[$i]}
  # теперь проверяем лог и отсылаем письмо
```

- Надо обратить внимание:

`${logPaths[@]}` все элементы массива.
`${!logPaths[@]}` все его индексы.

- Также возможный синтаксис инициализации: `arr=( $(ls) )`

Функции в bash

- Простое сложение.

```
#!/bin/bash

function myfunc {
  local value=$(( $1 + $2 ))
  echo $value
}

value=100
myfunc 10 15

echo $value

$ ./add.sh
```

Ограничения bash

- Любой shell scripting язык является **плохим** скриптовым языком.
- Кроме того у bash проблемы с переносимостью.
- Но недостатки bash это и его достоинства.