Конструкции.

Функции

Вызов функций похож на запуск программ.

Как и программы, функции возвращают код возврата (целое число).

Код возврата `0` означает, что всё хорошо; любое другое — нет

Передать аргументы в функцию можно, как и в программу, с помощью аргументов.

Функция может что-то писать в файловые дескрипторы.

Функции

```
sum_and_print() {
    printf $(($1 + $2))
}

sum_and_print 10 5 | wc -c #?
```

Job control

- `^Z` (Ctrl+Z) — отправить `SIGSTOP`. Процесс остановит работу и вернуть контроль.
- `fg` — возобновить остановленную задачу
- `bg` — выполнять задачу в фоновом режиме
- `jobs` — посмотреть текущие задачи

Job control

```
$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=103 time=7.188 ms

# fg %1
# fg 1
$ fg ping
64 bytes from 8.8.8.8: icmp_seq=1 ttl=103 time=6.860 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=103 time=6.430 ms
```

Subshell

У shell имеется некоторое состояние: набор значений переменных, то, какая директория является текущей, состояние переменных окружения и так далее.

Можно запустить ряд команд так, чтобы любые изменения окружения, которые в них произойдут, не были видны снаружи:

Subshell

```
$ cd /etc

# дочерний процесс?
$ (cd /; a=5; b=6) # запустить всё в subshell

$ pwd
/etc

$ echo $a
bash: a: unbound variable
```

Subshell

Subshell целиком можно отправить исполняться в фоне, а также всему subshell можно перенаправить файловые дескрипторы:

```
$ (ls; ls) > /tmp/t.txt
$ cat /tmp/t.txt
dotfiles/
unix.md
Downloads/
dotfiles/
unix.md
Downloads/
```

Переменные VS переменные окружения

Есть переменные:

```
$ a=x # переменная shell
```

Есть переменные окружения:

```
$ PATH="$HOME/.local/bin:$PATH"
```

Переменные окружения наследуются дочерними процессами (копируются в момент запуска).

Переменные VS переменные окружения

```
$ sh -c "echo $PATH"
/usr/local/sbin:/usr/local/bin:/usr/bin:/usr/bin/site_perl:/usr
/bin/vendor_perl:/usr/bin/core_perl

$ PATH=/bin

$ sh -c "echo $PATH"
/bin
```

Переменные VS переменные окружения

`export` делает переменную переменной окружения

```
$ myvar=x
$ sh -c 'echo a${myvar}b'
ab
$ export myvar
$ sh -c 'echo a${myvar}b'
axb
```

Можно установить переменные для одной команды так:

```
$ myvar1=x myvar2=y sh -c 'echo $myvar1 $myvar2'
x y z
```

Команда .

`.` говорит, что скрипт нужно запустить в том же shell, в котором сейчас происходит работа.

```
$ echo cd / > script.sh
$ . script.sh
$ pwd
/
```

Конструкции. If

```
# если команда вернет 0, то выполнит тело
if команда
then
  тело
fi

# если команда вернет 0, то выполнит тело иначе альтернативное тело
if команда
then
  тело
else
  альтернативное тело
fi
```

Конструкции. for

```
for имя_переменной in набор значений
do
  тело цикла
done
```

Тело цикла выполнится столько раз, сколько значений в наборе, и внутри тела переменная `$имя_переменной` будет установлена в соответствующее значение.

Конструкции. for

```
$ for i in 1 2 3; do echo $i; done
1
2
3

$ for i in *; do printf "%s\n" "$i"; done # аналог ls для
бедных

$ for i in $(seq 0 10); do echo $i; echo $((10-i)); done
```

While

```
# пока команда не вернёт код ошибки != 0
while команда
do
  тело цикла
done
```

Например

```
while ! test -f x; do sleep 1; done # ждём, пока не появится файл `x`
```

test

`test(1)` — команда, с помощью которой можно проверять различные условия.

Например, существует ли какой-то файл, не директория ли это, равны ли две строки и так далее.

У этой команды есть синоним `[` с дополнительным требованием, что вызов `[` нужно завершать аргументом `]`.

test

```
$ [ 1 = 2
[: missing `]

$ test 1 = 2
$ echo $?
1

$ [ 1 = 1 ]
$ echo $?
0

$ [1=2]
-bash: [1=2]: command not found
```

read

`read` принимает на вход имена переменных и устанавливает эти переменные в значения, прочитанные с одной строки стандартного ввода:

```
$ printf "1 2\n3 4\n" | while read a b; do
>   echo "$a;$b"
> done

1;2
3;4
```

Что может пойти не так?

Программы не понимают, где подразумевался ключ, а где аргумент

Рассмотрим такую простую программу на Си:

```c
#include "stdio.h"

int main(int argc, char* argv[])
{
    for (int i = 0; i < argc; ++i) {
        printf("arg %d is %s\n", i, argv[i]);
    }
    return 0;
}
```

Что может пойти не так?

Запустим программу с предыдущего слайда:

```
# ./a.out -v -e x.sh y.txt "aha aha"
arg 0 is ./a.out
arg 1 is -v
arg 2 is -e
arg 3 is x.sh
arg 4 is y.txt
arg 5 is aha aha
#
```

Что может пойти не так?

Если захотеть переименовать файл `x` в `-i`, случится такое:

```
$ mv x -i
mv: missing destination file operand after 'x'
Try 'mv --help' for more information.
```

`mv` воспринял `-i` как флаг, решение:

```
$ mv x ./-i
```

Многие утилиты поддерживают специальный ключ `--`.

Смысл в том, что все аргументы после него не будут восприниматься как ключи.

```
$ mv -- -v -i # переместить файл -v в -i
```

Что может пойти не так?

Будьте аккуратны с раскрытиями, которые делает bash.

```
$ ls
-rf  important_file very_important_one
dir_with_important_files

$ rm * # ?
```

shell раскроет `*` в имена файлов текущей директории.

Поскольку утилиты все еще не знают какие аргументы передаются, `-rf` будет интерпретировано командой не как имя файла, а как ключ и призыв к действию.

printf vs echo

`echo` пользуются довольно часто там, где это делать опасно.

```
$ var=-n
$ echo "$var" # не выведет ничего
```

Выводить значения переменных нужно часто, и для этого есть хорошее безопасное решение:

```
printf "%s" "$var"
```