

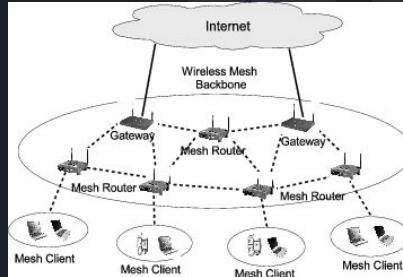
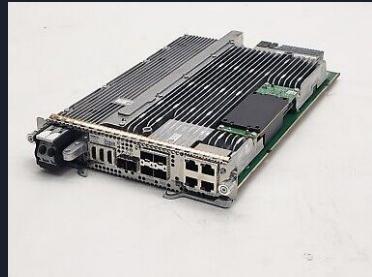
Bash Scripting for Automation

Karol
Przybylski

NobleProg

\$ whoami

Senior Software Engineer @ consult.red
Tech: Embedded Linux, Bash, C, C++





Uczestnicy

Czym się zajmuję?

Jaki jest dla mnie cel szkolenia?

Co aktualnie wiem o temacie szkolenia?

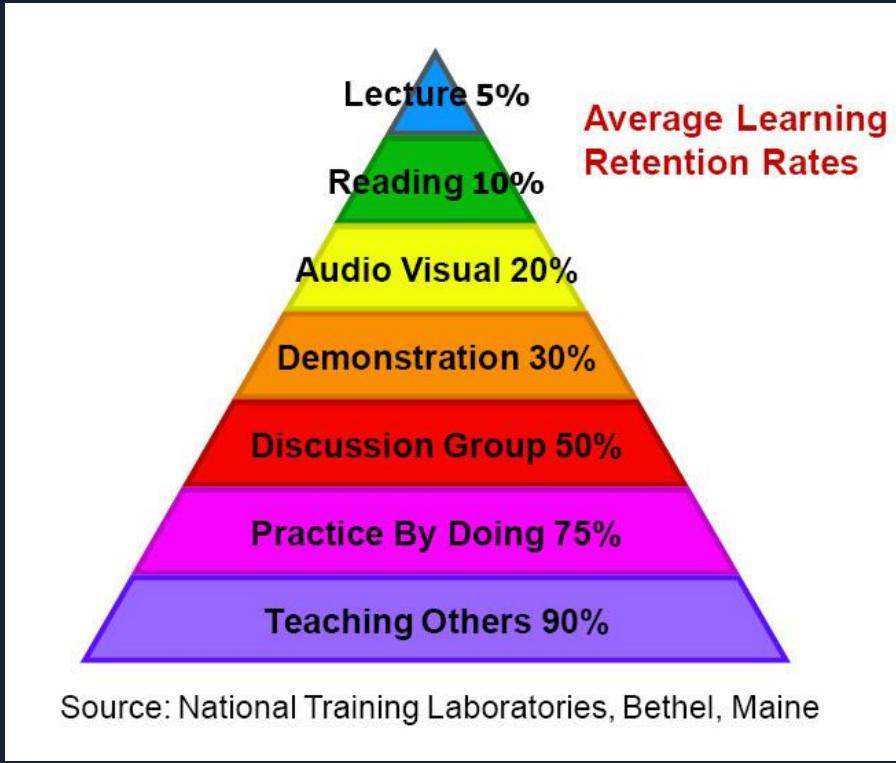
Informacje organizacyjne

Godziny: 9-16

Przerwa 1: ~10.30

Obiad: ?

Przerwa 2: ?



Agenda

Wprowadzenie, zrozumienie Bash - ogólnie

Pierwsze kroki

Tworzenie i uruchamianie Bash skryptów

Bash Scripting for Automation - automatyzacja

Integracja Bash z innymi narzędziami

Rozwiązywanie problemów - debugowanie

Podsumowanie i kolejne kroki - najlepsze praktyki



Bash - podstawowe informacje



Bash to język skryptowy do obsługi UNIX Shell

Język skryptowy vs komplikowany - interpreter

UNIX vs Linux ("Linux" -> Linux kernel+GNU)

Shell - podstawowe narzędzie do porozumiewania się z systemem



Kiedy używać Basha?

Automatyzacja

Skrypty administracyjne, interakcja z systemem

Umiarkowanie skomplikowane programy

Kiedy **nie** używać:

High performance/computation - C,C++

Skomplikowane i duże projekty - Python



Bash a inne języki skryptowe



- + Łatwo zacząć
- + Idealna integracja z Linuxem
- + Istniejący software
- Składnia potrafi się zrobić trudna
- Źle się skaluje

Idealny pod: zarządzanie/monitorowanie systemu, proste operacje na tekście, operacje na plikach, zadania administracyjne

Performance - szybszy przy prostych programach, jak idziemy w optymalizację to python może być szybszy



- Większy boilerplate
- + Świetnie się skaluje
- + Lepszy error handling

W zadaniach administracyjnych często po prostu wołamy w pythonie funkcje z basha...

(finaliści: perl, nodejs, windows: powershell)



Instalacja Basha na Linuxie

Zawsze będzie a jeśli nie to będzie SH - po prostu musimy mieć jak rozmawiać z kernelem

Q: Czy da się usunąć Basha?

Kod źródłowy: <https://github.com/bminor/bash>

Aktualizacja? Da się!

```
$ apt-get install --only-upgrade bash # $ -> oznacza komendę
```



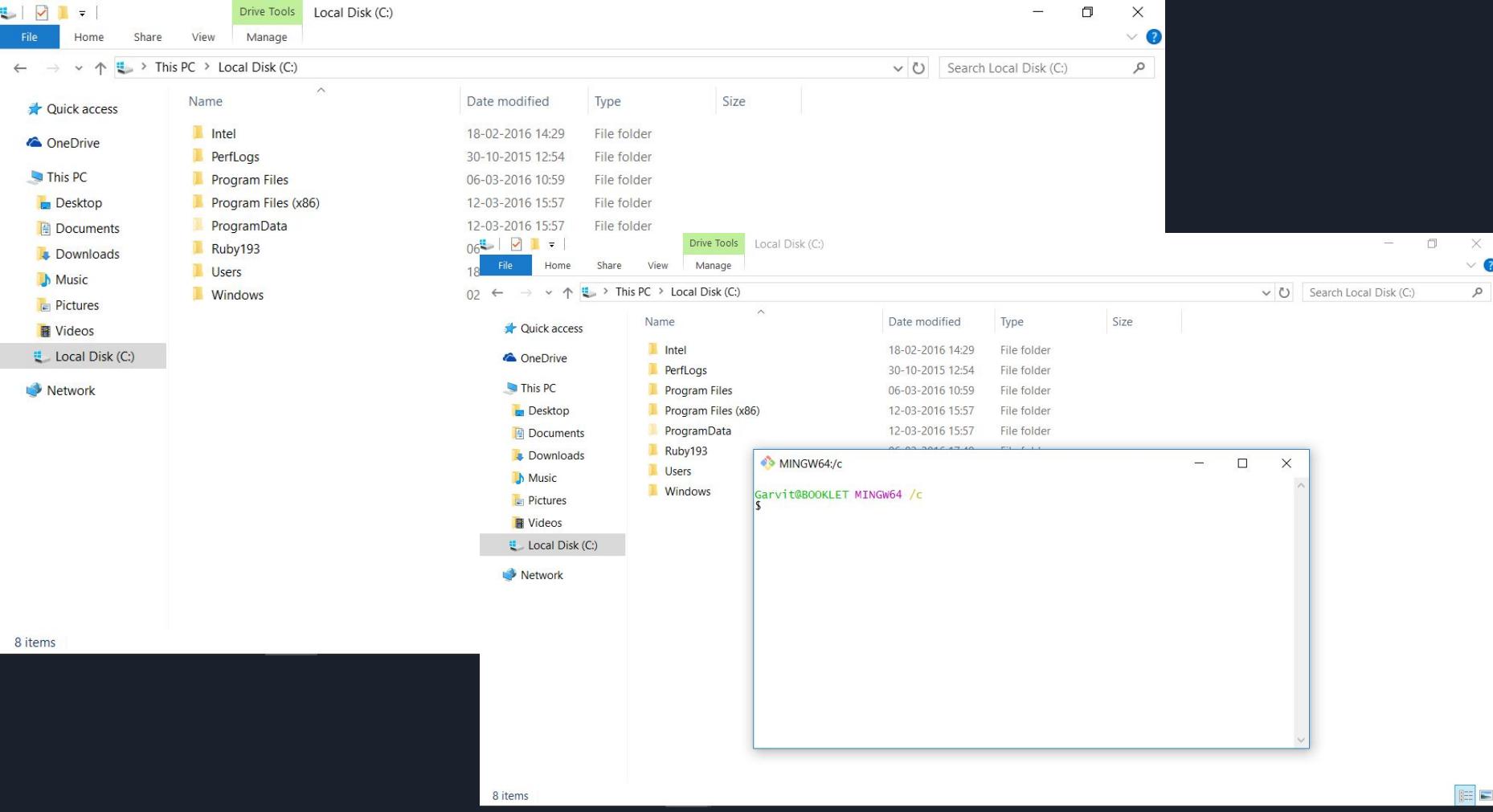
Instalacja basha na Windows

3 metody:

- Git bash <https://git-scm.com/downloads>
- Cygwin <https://www.cygwin.com/>
- Microsoft loves linux - WSL <https://learn.microsoft.com/en-us/windows/wsl/about>
(najlepsza opcja)

Kiedy może się przydać?

Gdy mamy skrypt bashowy i musi działać na Windowsie i Linuxie (wtedy WSL). Warto również pomyśleć o pythonie



Pierwsze programy



The image shows a laptop screen displaying a code editor with PHP code. The code is for a registration form, specifically for the 'extended-registration' plugin. The code includes logic for handling user input, validating fields like email and password, and setting session variables. The code editor interface shows file navigation, code snippets, and a toolbar at the bottom.

```
functions.php
1 // Include $view_path . 'header.php';
2
3 foreach ($fields as $field) {
4     er_render_field($field);
5 }
6
7 include($view_path . 'footer.php');
8
9 function er_render_registration_form() {
10     $results = array('errors' => array(),
11                     'username' => null,
12                     'password' => null,
13                     'password_confirmation' => null);
14
15     $registration = ER_Model::factory('Registration');
16     $registration->time = date('Y-m-d H:i:s');
17
18     $fields = ER_Model::factory('Field')->loadTemplates();
19     foreach ($fields as $field) {
20         $field->template_id = $field['id'];
21         $field['id'] = null;
22
23         # Assign value and validate
24         switch ($field['type']) {
25             case 'title':
26             case 'description':
27                 continue;
28             break;
29
30             case 'checkbox':
31                 if ($field['value'] == isset($_POST[$field['unique_name']])))
32                     if ($field['required'] && !$field['value'])
33                         $results['errors'][$field['unique_name']] = 'Vous devez cocher cette case pour continuer.';
34                 break;
35
36             case 'email':
37                 $field['value'] = safe_get($_POST, $field['unique_name']);
38                 if ($field['required'] && !filter_var($field['value'], FILTER_VALIDATE_EMAIL))
39                     $results['errors'][$field['unique_name']] = 'Vous devez rentrer un email valide.';
40                 elseif (!filter_var($field['value'], FILTER_VALIDATE_EMAIL) == false)
41                     $results['errors'][$field['unique_name']] = 'Vous devez entrer une adresse courriel valide.';
42                 break;
43
44             case 'password':
45         }
46     }
47 }
```

Pierwszy program - podstawowy workflow

```
$ touch hello.sh  
$ nano hello.sh # modyfikacja pliku
```

```
#!/bin/bash  
  
echo "Hello World"  
  
exit 0
```

```
$ chmod +x hello.sh  
$ ./hello.sh # albo bash hello.sh
```



Exit codes

Exit code programu - informuje o tym z jakim wynikiem zakończył się program/polecenie

Q: Co oznacza exit code 0 (zero)?

Przykładowo, komenda \$ ls:

```
Exit status:  
 0      if OK,  
 1      if minor problems (e.g., cannot access subdirectory),  
 2      if serious trouble (e.g., cannot access command-line argument).
```



Ćwiczenie

Praca na DaDesktop

exercises/1_hello_world

Zamień shebang z pierwszej linijki na inny program ze ścieżki bin:

/bin/cat

/bin/touch

Co się dzieje i dlaczego?



Konfiguracja środowiska

Powszechnie “uznane” środowiska:

- **Vim/vi**
 - Zalety: działa wszędzie, ogromne możliwości, dostosowuje się do usera
 - Wady: vim
- **Visual Studio Code**
 - Lekki, łatwy do dostosowania, UI/UX
- **JetBrains IDE**
 - Płatny, w zależności od wersji; cięższy
- **Eclipse, emacs, nano, gedit**
- **Shellcheck**



Repozytorium kursu

https://github.com/KarolPWr/automation_bash

```
$ git clone https://github.com/KarolPWr/automation\_bash.git
```

```
$ cd automation_bash
```

```
$ ls
```

Porada: możecie zrobić fork na repo i na bieżąco commitować na swojego githuba

Ćwiczenie

Otwórz VS Code

Odpal poprzedni skrypt przez terminal

Sprawdź działanie shellcheck (extensions)

Terminal w VS code: ctrl+` (tam gdzie tylda)

Odpalenie vs code z ręki:

```
$ code -n .
```



Referencja: VS code bash integration

Run -> Add configuration -> Bash debug -> podmień launch.json (z folderu resources w repozytorium kursu)

Ponownie Run

Na środku pojawi się bardzo małe menu z możliwością odpalania skryptu strzałką

--- Do odpalenia na swoim komputerze ---



Zmienne

Definiowanie zmiennych (składnia):

- Zmienne są definiowane przy użyciu składni: NAZWA_ZMIENNEJ=wartość

```
HELLO_STRING="Hello, World!"
```

- Nazwy zmiennych mogą zawierać wielkie i małe litery, cyfry, podkreślenia i cyfry

```
zMi_enn4="Hello, World!"
```

- Dobrą praktyką jest spójne użwanie wielkich/małych liter
- Spacje nie są dozwolone wokół znaku równości podczas przypisywania wartości.

```
HELLO_STRING = "Hello, World!"
```

- Wstępnie zdefiniowane słowa kluczowe, takie jak if, else itp. nie mogą być używane jako nazwy zmiennych.
- Muszą zacząć się od _ (podkreślnik) lub litery, nie mogą od liczby
- Nie może być w nazwie spacji
- Są case-sensitive - \$myVar to coś innego niż \$MYVAR (lepiej trzymać się jednego sposobu)



Uzyskiwanie dostępu do wartości zmiennych

- Aby uzyskać dostęp do wartości zmiennej, użyj \$ZMIENNA

`$HELLO_STRING`

- Przykład: echo \$NAME wyświetli wartość zmiennej NAME

`echo $HELLO_STRING`

Zmiana wartości zmiennej

- Aby zmienić wartość istniejącej zmiennej, po prostu przypisz ją ponownie

`$HELLO_STRING=28`

Q: Czy miękkie typowanie to super pomysł czy niekoniecznie?



Ćwiczenie

Zdefiniuj zmienną w terminalu a następnie ją odczytaj

Q: Czy zmienna będzie widoczna w innej instancji terminala?



Typy zmiennych

Zdefiniowane przez użytkownika:

Łańcuchy tekstowe, czyli stringi, np. hello_string="Hello"

Liczby całkowite czyli integery np. hello_string=28

Tablice itp.

Tak naprawdę to wszystko jest stringiem

Zmienne systemowe - przykłady:

\$USER, \$HOME, \$SHELL

\$0 - nazwa skryptu

\$? - exit code ostatniej wywołanej komendy



Ćwiczenie

Sprawdź w terminalu jaki efekt da wyświetlenie zmiennych:

\$USER

\$HOME

\$SECONDS

\$0

Bonus: Jak wyświetlić wszystkie zmienne w naszej sesji?



Zmienne jako output z komend, czyli zastępowanie poleceń (command substitution)

Dane wyjściowe polecenia można przechwycić i przypisać do zmiennej za pomocą podstawiania poleceń:

```
TODAY=$ (date)  
echo "Today's date is $TODAY"
```

Cytowanie zmiennych

Gdy zmienna zawiera spacje lub znaki specjalne, musi być cytowana (najlepiej ZAWSZE cytować zmienne)

Pojedynczy cudzysłów - " - zachowuje dosłowną zawartość

Podwójny cudzysłów - "" - rozwija znaki specjalne (np. \$, `,\`)

```
NAME="JAN KOWALSKI"  
echo "$NAME" # Wyświetli się: JAN KOWALSKI
```



Ćwiczenie

Małe eksperymenty z cytowaniem

Co wydarzy się jeśli zamienimy "" na " w komendzie echo
ze zmienną?

```
NAME="JAN KOWALSKI"  
echo "$NAME"
```



Ćwiczenie

Napisz skrypt, zadeklaruj w nim 4 zmienne:

- Imię
- Nazwisko
- Wiek
- Ulubione jedzenie
- (informacje nie muszą być prawdziwe)

Następnie wyświetl na terminalu:

Hello, my name is <Imię nazwisko>

I'm <wiek> years old

I will be <wiek za rok> next year

My fav food is: <jedzenie>

P.S. Zobacz kolejny slajd



Referencja: matematyka w Bashu

```
$(( expression ))
```

```
num1=5
```

```
num2=10
```

```
$(( num1 + num2 ))
```

```
-----
```

Dla liczb dziesiętnych - \$ bc

```
echo "20/3" | bc -l
```



Korzystanie z \$ export

Zmienne mogą być definiowane globalnie przez polecenie \$ export

Sprawdź: \$ cat ~/.bashrc

Przetestuj scenariusz:

Zakomentuj zmienną w skrypcie a następnie zdefiniuj ją przez \$ export w terminalu

\$ export NAME="XXX"

“Exported variables get passed on to child processes, not-exported variables do not.”



Obsługa command line arguments



Argumenty linii komend

\$0	Nazwa skryptu
\$1	Argument pozycyjny
\${12}	Argument pozycyjny >9
\$#	Tzw. Argument count
\$*	Odnośnik do wszystkich argumentów

```
./hello_world.sh jan kowalski
```

```
$0: hello_world.sh
$1: jan
$#: 2
$*: jan kowalski
```



Ćwiczenie

Skopiuj poprzedni skrypt, i zmodyfikuj go tak aby zmienne były przesyłane przez linię komend a nie zapisane “na twardo” w skrypcie.

Wypisz także liczbę argumentów oraz nazwę skryptu.

Przykładowy output skryptu:

Hello from <NAZWA>!

First argument: <IMIE NAZWISKO>

Number of arguments: <LICZBA ARGUMENTOW>

All arguments: <wszystkie przesłane argumenty>

Ćwiczenie - c.d.

Napisz prosty “kalkulator” używając argumentów pozycyjnych i poznanej wcześniej arytmetyce zmiennych.

Wywołanie programu:

```
./calc.sh liczba1 liczba2 operacja
```

Np.

```
./calc.sh 33 12 +
```

Output:

45

Obsługiwane operacje: dodawanie i odejmowanie



Ważny wniosek

Nawet jeśli program wydaje się działać dobrze w kilku przypadkach, może być całkowitą patologią



Wartości domyślne zmiennych

Co jeśli polegamy na zmiennej, która nie została ustawiona?

Składnia:

```
FOO="${VARIABLE:-default}"
```

Efekt:

Do FOO zostanie przypisana wartość default jeśli VARIABLE nie jest ustawione (lub jest null)



Ćwiczenie

Dodaj do poprzedniego skryptu wartość domyślną dla wybranej zmiennej.

Przetestuj jej działanie. Możesz też skorzystać z:

```
$ unset <VARIABLE>
```



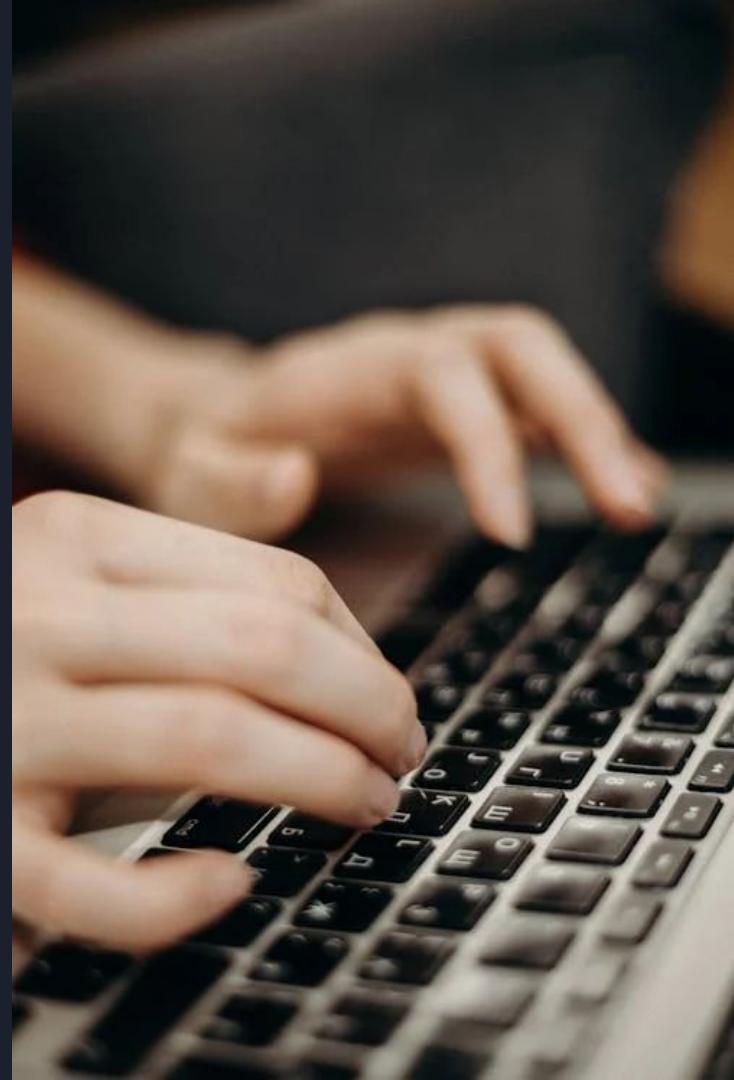
Skrypty interaktywne



User input

Metody:

- **Read, czyli analiza na bieżąco**
- Argumenty pozycyjne
- Zmienne środowiskowe





Polecenie \$ read

Jak skorzystać z funkcji read?

Czyli, jak korzystać z dokumentacji?

\$ man <program> - ogólnie dla programów

\$ man bash - pomoc bashowa

\$ man -k <program> - szukaj we wszystkich rozdziałach

\$ help <POLECENIE>

Ćwiczenie

Napisz program wpisujący zdefiniowaną przez użytkownika liczbę losowych znaków do pliku.

Zapisz tą wartość do zmiennej i następnie wyświetl w terminalu.

Przykładowy output programu:

Wpisz liczbę znaków do wpisania:

>użytkownik wpisuje liczbę<

Wpisano: 10

Na końcu programu dodaj prompt:

Naciśnij dowolny klawisz aby zakończyć program...

Po naciśnięciu klawisza program powinien się zakończyć.



Skrypty interaktywne są używane rzadziej niż argumenty pozycyjne - dlaczego?

Używane metody posortowane po popularności:

1. Argumenty pozycyjne
2. Zmiennie środowiskowe
3. Read



Debugowanie - podstawy





Najważniejsze metody

Patrzenie w kod

“Gumowa kaczka” i rozmowa z kimś innym

```
$ set -x LUB $ bash -x <SCRIPT>
```

Wyświetlanie każdej wykonywanej operacji

Działa też w terminalu

Jak wyłączyć?

Ustawianie breakpointów przez \$ exit

Q: \$ set -e -v -n # jak się dowiedzieć co to?

```
$ help set
```



Ćwiczenie

Zdebuguj skrypt w rozdziale 5_ez_debug

Użyj:

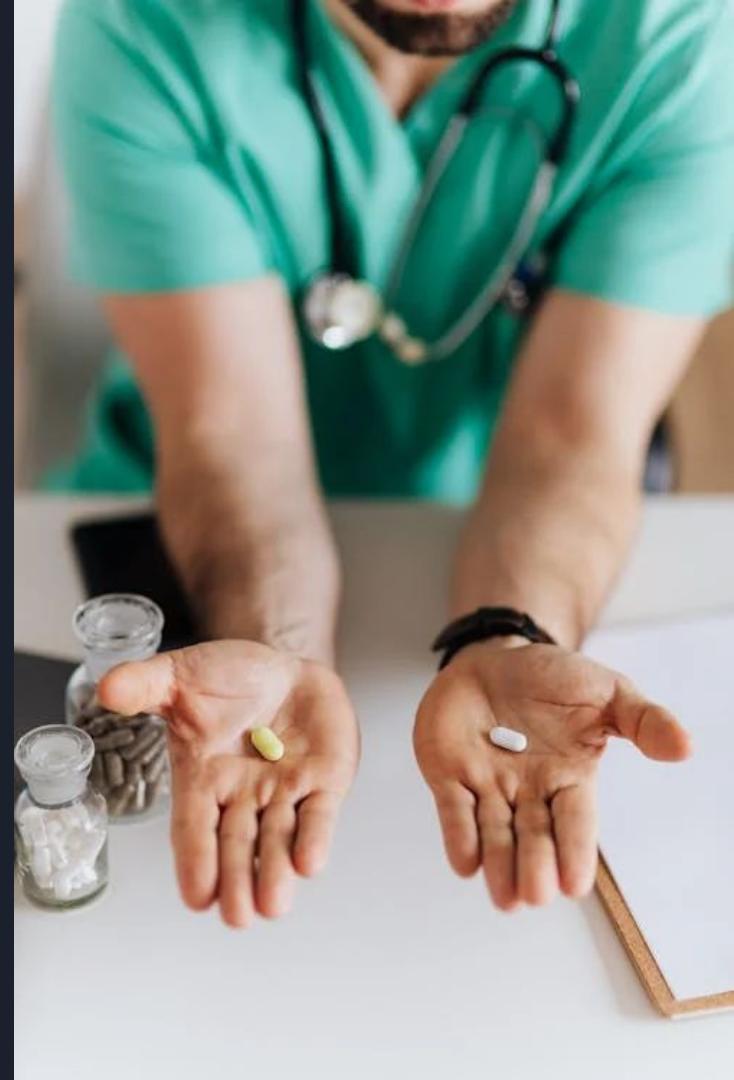
```
$ set -x lub bash -x
```

Zapoznaj się z \$ man set. Jakie inne opcje mogłyby być przydatne w debugowaniu?



Wyrażenia warunkowe

Jeżeli X to wykonujemy Y





Wyrażenia warunkowe



Do korzystania z instrukcji warunkowych w skryptach Bash wykorzystuje się przede wszystkim konstrukcje if, else i elif.

Umożliwiają one wykonywanie określonych bloków kodu w oparciu o określone warunki.

Konstrukcja if/elif/else

```
if condition; then  
    statement  
    ...  
fi
```



```
if condition; then  
    statement  
    ...  
elif condition2; then  
    statement  
    ...  
else  
    statement  
    ...  
fi
```



Składnia - if

```
if condition; then  
    statement  
    ...  
fi
```

VAR=5

```
if [ "$VAR" -gt 3 ]; then  
    echo "The number is greater than 3"  
fi
```

Tabulacje?

Odstęp między []



Czym jest []?

[] to skrót dla wyrażenia \$ test

\$ man test

\$ test jest shell built-in'em - występuje pod postacią [EXPRESSION] dla wygody i czytelności

Testowanie łańcuchów tekstowych (stringów)

Testowanie liczb całkowitych (integerów)

Testowanie plików



Operatory

-eq: equal to

-ne: not equal to

-gt: greater than

-lt: less than

-ge: greater than or equal to

-le: less than or equal to

Wszystkie operatory znajdziesz w \$ man test



Dlaczego “normalne” operatory nie są wspierane

Normalne operatory: <, >, = itp.

Operator > jest już zarezerwowany, jako przekierowanie strumienia (np. Do pliku)

Wyrażenie w [] (czyli \$ test) zostanie zinterpretowane jako przekierowanie

Lepsza alternatywa - [[]]

Bash wprowadza wsparcie dla nowego operatora warunkowego, czyli [[]]

[[]] wspiera porównywanie regexami

```
str="hello123"
if [[ $str =~ ^hello[0-9]+$ ]]; then
    echo "Pattern matched!"
fi
```

[[]] pozwala stosować operatory logiczne w środku nawisów

```
if [[ -f file.txt && -r file.txt ]]; then
    echo "file.txt exists and is readable."
fi
```

Bezpieczne i przewidywalne porównywanie stringów

```
var="a string with spaces"
if [[ $var == "a string with spaces" ]]; then
    echo "Strings match!"
fi
```



[] vs [[]]

Bardziej przewidywalne, spójne zachowanie

```
file=""  
if [[ -f $file ]]; then  
    echo "This won't cause an error, even if file is empty."  
fi
```

Do zapamiętania:

[] - gdy zależy nam na kompatybilności

[[]] - w każdym innym wypadku



Operatory warunkowe

[

Standard POSIX

[[

Dostępny w ‘ulepszeniach’ Bourne Shell

Wyrażenia parsowane “dosłownie”

“Normalne” zachowanie operacji logicznych

Osobna komenda

Wspiera pattern matching i Regexp

Builtin expression

Mniej niespodzianek



Kompatybilność



Coś nie działa lub zachowuje się dziwne? Problemem nie musi być sam skrypt, a stosowany shell.

\$ sh -> wersja “pierwotna”, mniej ficzerów, np nie zawsze jest wsparcie dla [[]]

\$ ksh, zsh, ash ... -> warto pamiętać, a agenty AI nie zawsze ogarniają

elif

```
if condition; then  
    statement  
    ...  
elif condition2; then  
    statement  
    ...  
else  
    statement  
    ...  
fi
```

```
A=10  
B=20  
  
if [[ $A -gt $B ]]; then  
    echo "A is greater than B"  
elif [[ $A -lt $B ]]; then  
    echo "A is less than B"  
else  
    echo "A is equal to B"  
fi
```



Ćwiczenie

Napisz skrypt przyjmujący jeden argument pozycyjny (liczbę).

Jeśli argument jest pusty, zakończ skrypt.

Następnie sprawdź czy przekazany argument jest większy niż dana liczba (np. 5)

Zadbaj o odpowiednie komunikaty dla użytkownika



Łączanie warunków

```
if [[ -d /etc && -r /etc/passwd ]]; then
    echo "The /etc directory exists AND the /etc/passwd file is readable."
else
    echo "Either /etc directory does not exist OR /etc/passwd file is not readable."
fi
```

```
if [[ -f /etc/passwd && -r /etc/passwd || -f /etc/shadow && -r /etc/shadow ]]; then
    echo "Either /etc/passwd is a readable file, OR /etc/shadow is a readable file."
else
    echo "Neither /etc/passwd nor /etc/shadow are readable files."
fi
```



Ćwiczenie

Ala ma kota

Napisz program, który pobierze od usera liczbę kotów (między 1 a 100). Następnie wyświetli na ekranie:

Ala ma <LICZBA> kot<PRZYROSTEK>

Ważne: zdanie musi być poprawne gramatycznie!

Np.

Ala ma 27 kotów

Bonus: Napisz program testujący poprawność programu
Ala ma kota



Ćwiczenie

Skrypt tworzący kopię zapasową - repozytorium

Case statement

```
case expression in
    case1)
        statement
        ...
    ;;
    case2)
        statement
        ...
    ;;
    *)
        statement
        ...
    ;;
esac
```

```
echo "Please enter a number between 1 and 3:"
read -r choice

case $choice in
    1)
        echo "You chose option 1."
    ;;
    2)
        echo "You chose option 2."
    ;;
    3)
        echo "You chose option 3."
    ;;
    *)
        echo "Invalid choice."
    ;;
esac
```



Ćwiczenie

Zmodyfikuj poprzedni skrypt. Daj użytkownikowi możliwość spakowania plików różnymi programami:

- 1) tar
- 2) gzip
- 3) Xz

Możliwa implementacja: przekazanie argumentu pozycyjnego do struktury case

Exit codes w praktyce

```
$ command1 || command2  
cat not_existing || ./hello_world.sh # wynik?
```

```
$ command1 && command2  
./hello_world.sh && cat not_existing # wynik?
```

```
$ hello.sh
```

```
$ echo $?
```

```
ls /path/to/file(FILE && cp FILE /new/path/FILE
```



Kolorowy terminal



`$()` versus `${}`

`$()` - command substitution, służy do posługiwania się outputem komend

`${}` - parameter expansion, do manipulowania zmiennymi i modyfikowania ich

```
name="Alice"
```

```
echo "Hello, ${name}!"
```

```
string="Hello"
```

```
echo "Length: ${#string}" # wyświetli 5
```



Ćwiczenie

Zmodyfikuj output z dowolnego poprzedniego ćwiczenia w dowolny sposób

Np. error na czerwono, notice na żółto

Wypróbuj program \$ lolcat

```
ls -la | lolcat - as 25
total 150
drwxrwxr-x  7 karol.przybylski karol.przybylski  18 wrz  7  2023 .
drwxr-xr-x 20 karol.przybylski karol.przybylski  23 sie  1 11:26 ..
-rwxrwxr-x  1 karol.przybylski karol.przybylski 2044 wrz  7  2023 buildall.sh
-rwxrwxr-x  1 karol.przybylski karol.przybylski 2382 wrz  7  2023 build_container.sh
-rw-rw-r--  1 karol.przybylski karol.przybylski 2582 wrz  7  2023 build-install-dumb-init.sh
-rw-rw-r--  1 karol.przybylski karol.przybylski 18092 wrz  7  2023 COPYING
-rwxrwxr-x  1 karol.przybylski karol.przybylski 1270 wrz  7  2023 deploy.sh
-rwxrwxr-x  1 karol.przybylski karol.przybylski 722 wrz  7  2023 distro-entry.sh
drwxrwxr-x  9 karol.przybylski karol.przybylski   9 wrz  7  2023 dockerfiles
drwxrwxr-x  8 karol.przybylski karol.przybylski  13 wrz  7  2023 .git
drwxrwxr-x  3 karol.przybylski karol.przybylski   3 wrz  7  2023 .github
-rw-rw-r--  1 karol.przybylski karol.przybylski  65 wrz  7  2023 .gitignore
drwxrwxr-x  2 karol.przybylski karol.przybylski   3 wrz  7  2023 helpers
-rw-rw-r--  1 karol.przybylski karol.przybylski  894 wrz  7  2023 install-buildtools-make.sh
-rw-rw-r--  1 karol.przybylski karol.przybylski  861 wrz  7  2023 install-buildtools.sh
-rw-rw-r--  1 karol.przybylski karol.przybylski 18092 wrz  7  2023 LICENSE
-rw-rw-r--  1 karol.przybylski karol.przybylski  15 wrz  7  2023 README.md
drwxrwxr-x  4 karol.przybylski karol.przybylski   4 wrz  7  2023 tests
lolcat: as: No such file or directory
```

Tablice (arrays)

G 1 D , 4 b Y % J H
m 10 N Ö E A s t i d
4 7 G Z , d X ~





Tablica - podstawowa struktura danych

Podobnie jak w innych językach, tablice w bashu służą do przechowywania zbiorów danych i razem z pętlami umożliwiają jeszcze większą automatyzację.

Przechowujemy wtedy **wiele różnych wartości pod jedną zmienną**.

Deklaracja:

```
arr=(element1 element2 element3)
```

Z użyciem declare:

```
declare -a arr=("element1" "element2" "element3")
```

```
$ echo ${my_array[1]}
```



Ćwiczenie

Stwórz tablicę z dowolnymi elementami, o długości >1

Używając argumentów linii komend dodaj 3 elementy do tablicy - na początek, na koniec i w środku tablicy.

Pętle

while | until | for



for

Podstawa: FOR a given list DO something

Składnia:

```
for element in [collection]
do
    statement
    ...
done
```

```
words=("apple" "banana" "cherry"
"date")
for word in "${words[@]}"; do
    echo "Fruit: $word"
done
```

One-liner:

```
for system in ubuntu windows mac;do echo $system;done
```



Generowanie sekwencji

W terminalu:

```
$ echo {0..10}
```

```
{START..END[..INCREMENT]}
```

Ćwiczenie: wyświetl liczby parzyste od 8 do 92

```
$ seq 120
```



Case study - sprawdzanie stanu serwisu

Piszemy skrypt sprawdzający czy serwis NGINX wstał i pracuje



Pętle w terminalu

Wyrażenia pętli: for, while etc. to tzw. Shell built-ins - możemy ich używać w terminalu do ułatwiania sobie codziennej pracy, bez potrzeby pisania skryptu (zaoszczędzimy kilka minut)

Np. Wykonaj działania na określonych folderach

```
$ dirs="/etc /home"

$ echo "$dirs"
/etc /home

$ for dir in $dirs; do echo $dir;done
/etc
/home
```



Ćwiczenie

Napisz one linera, który:

Dla zadanej liczby naturalnej (np. 120)

Odlicza liczbę sekund do końca przerwy

Na koniec wypisuje "Koniec przerwy"

Przykładowe wyjście:

› bash oneliner.sh

119 seconds remaining

118 seconds remaining

117 seconds remaining

116 seconds remaining



Control flow

Często występuje sytuacja, w której chcemy zakończyć pętlę wcześniej lub pominać niektóre elementy - wtedy przychodzą z pomocą instrukcje:

Break - natychmiastowe wyjście z pętli

Continue - 'continue to the top of the loop- tzn. Powrót do początku pętli (pomijamy resztę)

```
for element in "apple" "pear" "salad" "watermelon"
do
    if [[ $element == "salad" ]]; then
        echo "Salad found"
        break
    else
        continue
    fi
done
```

While i until

While - execute WHILE expression is TRUE

Until - execute UNTIL expression is TRUE

Składnia:

```
while [ some test ]
do
<commands>
done

until [ some test ]
do
<commands>
done
```

```
counter=1
```

```
while [[ $counter -le 5 ]]; do
echo "Counter: $counter"
((counter++))
done
```



Ćwiczenie

1: Sprawdź czy programy: ls, cat, nginx, lolcat są zainstalowane na systemie

Zadanie wykonaj z pomocą tablicy z listą programów do sprawdzenia



Ćwiczenie

Napisz grę “Zgadywanie liczby”

Program wybiera liczbę od 1 do 100

Użytkownik zgaduje jaka liczba została wybrana

Po każdej odpowiedzi użytkownik dostaje informację czy jego próba odgadnięcia jest większa czy mniejsza od tego co wymyślił program

Jeśli użytkownik zgadł - wyświetl komunikat i zakończ program



Ćwiczenie

Sortowanie plików

Mając następujące pliki w ścieżce ~/workspace/ex:

1.pdf 2.jpg 3.png 4.abcd

Posegreguj je biorąc pod uwagę extension i przenieś do nowych folderów. Czyli w ~/workspace/9_loops pojawią się foldery:

.pdf .jpg .png .abcd

A w folderach znajdą się pliki z odpowiednim rozszerzeniem



Demo

Debugowanie z VS code - wady i zalety

Praca z plikami





Czytanie danych z pliku

Podstawowa struktura:

```
while read -r line; do
    printf '%s\n' "$line"
done < "file"
```

Generowanie testowego pliku

```
curl http://metaphorpsum.com/sentences/3
```

```
curl http://metaphorpsum.com/paragraphs/20 > test.txt
```

Reuse kodu - przypomnienie



Ćwiczenie

Analiza danych z pliku

- Skrypt pobiera ścieżkę do pliku jako argument
- Obsługa błędu jeśli plik nie istnieje lub nie ma końcówki .log
- Skrypt przygotowuje statystyki:
 - Liczba log entries (linijek logu)
 - Ile ERROR
 - Ile INFO



Ćwiczenie

Analiza pliku konfiguracyjnego z journalctl

Obsługa zmiennej IFS



Budowanie interaktywnego menu

Nieskończone pętle - to зло absolutne

NIE - są szeroko wykorzystywane: systemy czasu rzeczywistego, grafika, menu

(choć zawsze zwrócią uwagę współpracowników...)



Ćwiczenie

Dopisz brakującą logikę do skryptu implementującego “wieczne” menu

Pętla while

Read na user input



Command PATH

System PATH - zawiera listę ścieżek, która jest przeszukiwana gdy wywołujemy program

```
$ echo $PATH && echo $PWD
```

```
$ mkdir -p $HOME/bin
```

```
$ export PATH=$PATH:$PWD
```



Cykl życia skryptu Bash

Q: Co się dzieje gdy uruchamiamy skrypt?

1. Załadowanie programu do pamięci RAM
2. Shell przeszukuje swój \$PATH by znaleźć program do odpalenia
3. fork() - kopia procesu
4. execve() - podmiana pamięci przez inny proces (np. Komenda top)
5. Parent process kontynuuje działanie, obserwuje swoje dzieci przez funkcję wait()

Dla dociekliwych: widać do dokładnie wywołując \$ strace <nazwa programu>



Ćwiczenie

Dodaj ścieżkę z dowolnym skryptem do PATH

Wykonaj ten skrypt z innej ścieżki

Porada: jak najczęściej używaj tab completion

Funkcje





Motywacja funkcyjna

- Lepsza czytelność
- Krótszy kod
- Łatwiejsze modyfikacje
- Dokumentacja (np. Nazwa funkcji)
- Modularyzacja, dzielenie się kodem



Podstawy

```
function_name () {  
    # Commands to execute  
}
```

Wywołanie

```
function_name
```

```
show_date_time () {  
    echo "Current date and time: $( date )"  
}
```

Zakres zmiennych a funkcje

W języku Bash zmienne są domyślnie globalne, niezależnie od tego, czy zostały zdefiniowane wewnątrz funkcji, czy poza nią.

Aby utworzyć zmienną lokalną wewnątrz funkcji, należy użyć słowa kluczowego local:

```
calculate_square () {  
    local NUMBER=55  
    echo "The square of $number is $((NUMBER * NUMBER))"  
}
```



Przesyłanie argumentów do funkcji

```
calculate_sum () {  
    local A=$1  
    local B=$2  
    local sum=$((A + B))  
    echo "The sum of $A and $B is $sum."  
}
```

```
calculate_sum 10 20
```



Zwracanie wartości?

Metody:

- `return` # tylko exit status
- Zmienne globalne
- Command substitution (tak jak normalne komendy)



Zwracanie wartości c.d.

```
calculate_sum() {  
    local A=$1  
    local B=$2  
    local sum=$((A + B))  
    #echo "The sum of $A and $B is $sum."  
    echo $((A+B))  
}  
  
val=$(calculate_sum 10 20)  
echo "Value: $val"
```



Korzystanie z funkcji z innych plików

```
$ source script.sh
```

```
$. script.sh
```

Nie wywołuje forka, nie podmienia pamięci

➤ help source

```
source: source filename [arguments]
```

Execute commands from a file in the current shell.



Ćwiczenie

Zaimplementuj generowanie losowych zdań/akapitów jako funkcję ogólnego dostępu

Stwórz plik utils.sh

Zaimplementuj generację zdań jako funkcję przyjmującą za argument liczbę zdań i paragrafów do wyprodukowania

Użyj stworzonej funkcji w innym pliku



Efektywne logowanie

Samo używanie \$ echo nie wystarczy - dlaczego?

Kiedy coś się dzieje?

Z czym mamy do czynienia? Info, Error, Debug?

Przykład - \$ journalctl



Ćwiczenie

Efektywne logowanie

Napisz funkcje ułatwiające logowanie w skrypcie.

`log_info()`

`log_error()`

Preferowany format:

`DATA LOG_LEVEL`

Zimportuj funkcje do innego skryptu i użyj

BONUS: Dodaj i przetestuj dodatkowo użycie zmiennej systemowej `$LINENO`



Przekierowanie do pliku

Jak zrealizować logowanie do pliku?

Pierwsza opcja - proste przekierowanie (w formie append)

```
echo "[ERROR] [$timestamp] $message" >> "$LOG_FILE"
```

Jak logować jednocześnie do stdout i do pliku?

```
echo "[ERROR] [$timestamp] $message" | tee -a "$LOG_FILE"
```



Przetwarzanie argumentów





Krótkie opcje

Do tej pory - tylko czyste argumenty pozycjne

Więcej kontroli - chcemy pozwolić userowi na używanie flag np.

```
./create_backups -d <DIRECTORY> -n <NAME>
```

Robienie tego ifami?

Rozwiązanie - getopt



Parsowanie krótkich opcji - getopt

Demo 13_argparse, 2_getopts



Ćwiczenie

Zmodyfikuj poprzedni program - lorem ipsum

Dodaj możliwość specyfikacji opcji za pomocą getopt

Np.

```
$ lorem ipsum -p <paragraphs|sentences> -n <number>
```

Dodaj IF który pozwoli przekazać tylko string 'paragraphs' lub 'sentences' do opcji -p

Komunikacja między skryptami





Rodzaje komunikacji

- 1) Message-based (wiadomości)
 - a) Queues (kolejki)
 - b) **Pipes** (potoki lub rurki)
 - c) Sockety (gniazda)
 - d) **Signals (sygnały)**
- 2) Pamięć współdzielona
- 3) API wyższego poziomu (abstrakcji)
 - a) Np D-BUS



Potoki

| -> bezpośrednie połączenie outputu jednej komendy z inputem drugiej

```
$ command1 | command2 # stdout -> stdin
```

Przepływ jest real-time, output jest natychmiast przesyłany do kolejnej komendy

```
$ ls -l | less
```

```
$ cat file.txt | wc -l
```

```
$ cat file.txt | sort | uniq
```

> vs | ? Przekierowanie kontra potok?

| -> programy

> -> strumienie, pliki

Zarządzanie procesami, sygnały





Zarządzanie procesami

Czym jest proces?

Monitorowanie - ps, ps -aux, pgrep, kill, top, htop

Kontrola procesów - kill (kolejny slajd)

Proces vs job?



Sygnały

```
$ man 7 signal
```

“Przerwanie softwarowe wskazujące programowi, że stało się coś ważnego”

Przerwanie (Interrupt) - zawieszenie wykonywania programu i obsługa przerwania

Przykłady - SIGSEGV

```
$ kill -l # lista wszystkich sygnałów
```

```
$ kill # wysyłanie sygnałów do programu
```



Wysyłanie sygnałów

```
$ kill -s SIGKILL <PID>
```

```
$ kill -9 <PID>
```

PID - Process Identifier

```
$ pidof firefox
```

Jak skutecznie zabić wszystkie instancje przeglądarki albo programu?

```
$ kill -s SIGKILL $(pidof firefox)
```



Najczęściej używane sygnały

SIGKILL

SIGSEGV

SIGILL

SIGINT # CTRL+C

SIGABRT, SIGTERM # subtelne różnice



Ćwiczenie

Otwórz dwie instancje terminala

Odpal proces sleep 500 w jednym

W drugim znajdź PID procesu i wyślij do niego sygnał

Podpowiedź:

```
$ ps -aux | grep -i sleep
```

```
$ kill -9 <PID>
```



Obsługa sygnałów

Co może się wydarzyć po otrzymaniu sygnału:

- Zakończenie procesu
- Zignorowanie sygnału
- Zatrzymanie procesu
- Core dump



Kontrola procesów w tle

Foreground, background processes

& - znak specjalny, powoduje wykonanie programu w tle

```
$ sleep 15 &
```

Jak zakończyć program?

```
$ echo $! # PID ostatniego programu który przeszedł do tła
```

```
$ kill -s SIGKILL $!
```

Q: Co zrobi `$ kill -0 <PID>`



Ćwiczenie

Napisz skrypt:

Odpalenie procesu w tle

Znajdź PID procesu

Monitoruj proces przez jakiś czas (np. Co 5 sekund przez minutę)

Zakończ proces po ustalonym czasie



Implementacja

:(){ :|:& };



Rozszyfrowanie

:(){ :|:& };

Definicja funkcji:

:(){ ... } definiuje funkcję o nazwie :

Wewnątrz funkcji znajdują się trzy części:

| (symbol potoku): tworzy nowy proces i wysyła do niego dane wyjściowe

: (nazwa funkcji): Rekursywnie wywołuje :function

& (operator tła): Uruchamia wywołanie rekurencyjne w tle

Wywołanie funkcji:

Po zdefiniowaniu funkcji, końcowe ; wykonuje funkcję.

Fork bomb

