Funkcje w skryptach powłoki

Poziom: podstawy **Dotyczy: bash 4+/5+** **Czas: ~15 min**

Funkcja w skrypcie to nazwany blok komend, do którego wracamy wielokrotnie bez przepisywania go za każdym razem. Ta strona pokazuje definicję, argumenty, zwracanie wartości i konwencję układu skryptu, którą trzymamy w całej tej książce.

Skróty użyte na tej stronie (1)

Skrót	Co znaczy
POSIX	Portable Operating System Interface: wspólny zestaw zasad, dzięki któremu polecenie działa tak samo na różnych Uniksach

Fiszka

TYLKO ODCZYT

Definicja i wywołanie funkcji

```
powitaj() {
    echo "Cześć, $1"
}

powitaj "adm_user"
```

Nawiasy `()` po nazwie są puste zawsze — to nie jest miejsce na listę parametrów, jak w innych językach. Argumenty funkcja odbiera dokładnie tak samo jak skrypt: przez `$1`, `$2`, `$@`.

Przykład:

```
Cześć, adm_user
```

TYLKO ODCZYT

Zmienna widoczna tylko wewnątrz funkcji: local

```
oblicz_sume() {
    local a="$1"
    local b="$2"
    local wynik=$((a + b))
    echo "$wynik"
}
```

Bez `local` zmienne `a`, `b` i `wynik` byłyby globalne dla całego skryptu i nadpisałyby każdą istniejącą zmienną o tej samej nazwie poza funkcją — mechanizm zasięgu opisujemy szerzej na stronie Zmienne, parametry i argumenty.

TYLKO ODCZYT

Zwrócenie statusu przez kod wyjścia: return

```
plik_istnieje() {
    [[ -f "$1" ]]
}

if plik_istnieje /etc/hosts; then
    echo "jest"
fi
```

Funkcja bez jawnego `return` kończy się kodem wyjścia ostatniej wykonanej w niej komendy — tu wynikiem testu `[[ -f ]]`. `if` używa tego kodu dokładnie tak samo jak przy zwykłej komendzie zewnętrznej. Jawny `return N` pozwala zwrócić konkretny kod niezależnie od tego, co wykonało się jako ostatnie.

TYLKO ODCZYT

Zwrócenie wartości przez standardowe wyjście: echo

```
nazwa_hosta_krotka() {
    echo "${1%%.*}"
}
```

```
wynik=$(nazwa_hosta_krotka "app01.zielona.example")
echo "$wynik"
```

Funkcje w bashu nie zwracają wartości tak jak w innych językach — `return` przyjmuje wyłącznie liczbę 0–255 jako kod wyjścia. Żeby „zwrócić" napis albo liczbę, funkcja wypisuje go na standardowe wyjście, a wywołujący łapie to podstawieniem komendy `$(...)`.

Przykład:

```
app01
```

TYLKO ODCZYT

`return` kontra `exit`

```
sprawdz_uprawnienia() {
    [[ $EUID -eq 0 ]] || return 1
    echo "jesteśmy rootem"
}

sprawdz_uprawnienia || echo "brak uprawnień roota, kontynuuję ograniczoną wersję"
```

`return` kończy tylko bieżącą funkcję i oddaje sterowanie do miejsca jej wywołania — skrypt leci dalej. `exit` kończy cały skrypt (albo, użyty przez pomyłkę w skrypcie wczytanym przez `source`, całą powłokę interaktywną). W funkcji sięgamy po `return`, `exit` zostawiamy dla `main` albo dla sytuacji, w których dalsze działanie skryptu nie ma sensu.

TYLKO ODCZYT

Biblioteka funkcji w osobnym pliku: source

```
# plik: biblioteka.sh
log() {
    printf '[%s] %s\n' "$(date '+%Y-%m-%d %H:%M:%S')" "$1"
}
```

```
# plik: skrypt.sh
#!/bin/bash
```

```
source "$(dirname "$0")/biblioteka.sh"


log "start skryptu"
```

`source` wczytuje definicje funkcji z osobnego pliku do bieżącej powłoki — dzięki temu ta sama funkcja `log` czy `die` działa identycznie we wszystkich skryptach zespołu, a poprawka trafia do jednego miejsca zamiast dziesięciu kopii. `dirname "$0"` gwarantuje, że biblioteka jest szukana względem lokalizacji skryptu, a nie względem katalogu, z którego akurat go uruchomiono.

printf = wypisz sformatowany tekst, ściślejsza odmiana echo · **source** = wykonaj plik w bieżącej powłoce zamiast w nowej, zmienne z niego zostają dostępne dalej

ZMIENIA SYSTEM

Szkielet skryptu z funkcjami: main na końcu

```
#!/bin/bash

usage() {
    echo "Użycie: $0 [-v] NAZWA_USLUGI" >&2
}

log() {
    printf '[%s] %s\n' "$(date '+%Y-%m-%d %H:%M:%S')" "$1"
}

die() {
    echo "BŁĄD: $1" >&2
    exit "${2:-1}"
}

main() {
    [[ $# -ge 1 ]] || { usage; exit 2; }
    local usluga="$1"
    log "sprawdzam usługę: $usluga"
    systemctl is-active --quiet "$usluga" || die "usługa $usluga nie działa" 1
    log "usługa $usluga działa poprawnie"
}
```

```
main "$@"
```

Funkcje pomocnicze (`usage`, `log`, `die`) definiujemy na górze pliku, a jedyne wywołanie na poziomie skryptu to `main "$@"` na samym końcu. Dzięki temu cały skrypt czyta się od góry do dołu jak spis treści, a właściwa logika jest zamknięta w jednej funkcji, którą łatwo przetestować w izolacji.

printf = wypisz sformatowany tekst, ściślejsza odmiana echo · **exit** = zakończ bieżącą powłokę albo skrypt z podanym kodem wyjścia · **systemctl** = system control: sterowanie usługami systemd

TYLKO ODCZYT

Wszystkie elementy funkcji

Element	Rola
`nazwa() { ... }`	definicja, nawiasy zawsze puste
`$1`, `$2`, `"$@"`	argumenty funkcji, tak jak argumenty skryptu
`local zmienna`	ograniczenie zasięgu do wnętrza funkcji
`return N`	kod wyjścia funkcji, 0–255, kończy tylko funkcję
`echo` / `printf` + `$(funkcja)`	„zwrócenie" wartości innej niż kod wyjścia
`source plik.sh`	wczytanie funkcji z zewnętrznego pliku

Uważaj

- `return` poza funkcją (albo poza skryptem wczytanym przez `source`) jest błędem składniowym w bashu.
- Funkcja wywołana jako warunek `if funkcja; then` nie przerywa skryptu nawet pod `set -e`, jeśli funkcja zawiedzie — to samo zachowanie, co przy zwykłej komendzie w warunku, opisane na stronie Tryb ścisły: set -euo pipefail.
- `local zmienna=$(polecenie)` maskuje kod wyjścia `polecenie` kodem wyjścia `local` — rozdzielamy deklarację i przypisanie na dwie linie, gdy kod wyjścia ma znaczenie.

Z życia wzięte

Z ŻYCIA WZIĘTE

Czwartek, 11:00. Tomek dopisuje do skryptu monitorującego na docker01 funkcję `policz()`, która wewnątrz używa pętli `for i in ...` bez `local`. Zewnętrzny licznik `i`, używany gdzie indziej w tym samym skrypcie do numerowania raportów, zostaje po cichu nadpisany, a kolejny raport wychodzi z błędnym numerem. Dodanie `local i` na początku funkcji kończy problem.

Po co to istnieje

Pojęcie: funkcja. Nazwany blok komend, wywoływany przez podanie jego nazwy, tak jak wywołuje się zwykłą komendę. Skrypt bez funkcji, który robi więcej niż kilkanaście linii, szybko zamienia się w ścianę tekstu, w której to samo sprawdzenie („czy usługa działa", „czy jesteśmy rootem") powtarza się w pięciu miejscach, skopiowane i wklejone, a poprawka błędu wymaga pamiętania o wszystkich pięciu. Funkcja daje nazwę powtarzalnej czynności i jedno miejsce do poprawki.

Analogia

Funkcja w skrypcie działa jak **formularz z gotowym nagłówkiem** w firmowym obiegu dokumentów.

Zamiast za każdym razem pisać od nowa, jakie dane trzeba podać i w jakiej kolejności, sięgamy po gotowy formularz „Wniosek o dostęp" — wypełniamy tylko pola zmienne (kto, do czego, na jak długo), a reszta procedury dzieje się tak samo za kulisami, niezależnie od tego, kto formularz wypełnił. Gdy firma zmienia procedurę, poprawia się jeden wzór formularza, a nie instruuje na nowo każdego, kto kiedykolwiek składał wniosek.

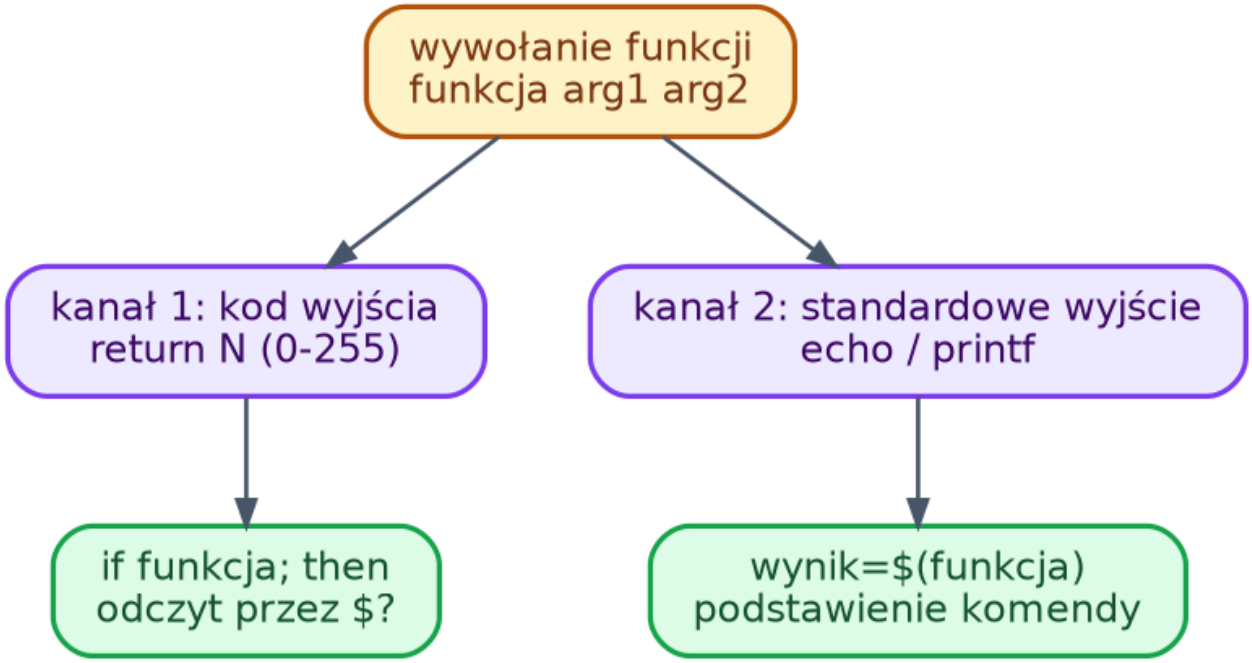
Ograniczenie tej analogii pokazuje realną pułapkę: formularz, wypełniony i złożony, nie zostawia żadnego trwałego wpisu w naszym własnym notatniku — chyba że sami przepiszemy sobie coś z kopii. Dokładnie tak działa `local` w funkcji: zmienna zadeklarowana wewnątrz znika, gdy funkcja kończy pracę, i nie zostawia śladu w reszcie skryptu, jeśli świadomie nie zwrócimy jej wartości przez `echo` albo nie pominiemy `local`, robiąc zmienną celowo globalną.

Jak to działa

Rozwiń: jak to działa pod spodem

Zasięg zmiennych w funkcji rozwijamy szczegółowo na stronie Zmienne, parametry i argumenty — tutaj liczy się jedna konsekwencja dla stylu pisania funkcji: każda zmienna pomocnicza, która nie ma świadomie zmieniać stanu globalnego skryptu, dostaje `local`. Odwrotna zasada, czyli zmienna globalna modyfikowana przez funkcję, jest czasem uzasadniona (na przykład licznik błędów zliczany przez wiele wywołań), ale wtedy zaznaczamy to komentarzem, bo dla czytającego kod jest to zaskoczenie.

Dwa kanały wyjścia funkcji. Funkcja ma do dyspozycji dokładnie dwa sposoby przekazania czegokolwiek na zewnątrz: kod wyjścia (`return`, liczba 0–255, odczytywana przez `$?` albo bezpośrednio w `if`) oraz standardowe wyjście (`echo`/`printf`, odczytywane przez podstawienie komendy `$(funkcja)`).



Funkcja ma dwa niezależne kanały: kod wyjścia czytany przez `if`/`$?` oraz standardowe wyjście przechwytywane przez `$(funkcja)`. (kliknięcie otwiera pełny rozmiar)

Nie da się „zwrócić" napisu przez `return` — próba `return "tekst"` jest błędem, bo `return` oczekuje liczby całkowitej w zakresie kodu wyjścia. Funkcje, które mają jednocześnie coś zwrócić i coś zgłosić (na przykład wynik obliczenia i informację o błędzie), rozdzielają te dwa kanały: wynik idzie na stdout, błąd na stderr, a status na `return`.

Biblioteka funkcji. Gdy ten sam zestaw funkcji (`log`, `die`, `require_root`) ma być używany w wielu skryptach na tym samym serwerze, trzymamy go w jednym pliku i wczytujemy przez `source` na początku każdego skryptu, który go potrzebuje. To jest dokładnie ten sam mechanizm, którym systemowe pliki startowe ładują funkcje powłoki interaktywnej — różnica jest tylko w tym, kto i kiedy woła `source`.

Konwencja układu skryptu. Funkcje na górze, `main "$@"` na dole, to nie estetyka — to pozwala uruchomić `bash -n` i zobaczyć całą strukturę skryptu bez czytania ciała każdej funkcji, oraz pozwala przetestować pojedynczą funkcję w izolacji przez `source skrypt.sh` i ręczne wywołanie jej z terminala, bez uruchamiania całego `main`. Ten sam wzorzec rozwijamy w pełnej, gotowej do skopiowania postaci na stronie Szkielet skryptu produkcyjnego.

Krok po kroku

1. Piszemy funkcję z walidacją argumentu i jawnym kodem błędu.

```
wymagaj_pliku() {
    local sciezka="$1"
    if [[ ! -f "$sciezka" ]]; then
        echo "BŁĄD: plik $sciezka nie istnieje" >&2
        return 1
    fi
}
```

2. Wywołujemy ją w warunku, bez sprawdzania `$?` ręcznie.

```
if wymagaj_pliku /etc/nieistniejacy.conf; then
    echo "plik jest, kontynuujemy"
else
    echo "pomijam ten krok"
fi
```

```
BŁĄD: plik /etc/nieistniejacy.conf nie istnieje
pomijam ten krok
```

3. Wyodrębniamy funkcję do biblioteki i wczytujemy ją w dwóch różnych skryptach.

```
mkdir -p /tmp/przyklad_bib
cat > /tmp/przyklad_bib/lib.sh <<'EOF'
log() { printf '[%s] %s\n' "$(date '+%H:%M:%S')" "$1"; }
EOF
cat > /tmp/przyklad_bib/a.sh <<'EOF'
#!/bin/bash
source "$(dirname "$0")/lib.sh"
log "skrypt a wystartował"
EOF
chmod +x /tmp/przyklad_bib/a.sh
/tmp/przyklad_bib/a.sh
```

Częste błędy

Próba zwrócenia napisu przez `return`.

```
pobierz_wersje() {
    return "5.14.0"
}
pobierz_wersje
```

```
skrypt.sh: line 2: return: 5.14.0: numeric argument required
```

`return` przyjmuje wyłącznie liczbę całkowitą 0–255. Naprawa: `echo "5.14.0"` w funkcji, `wersja=$(pobierz_wersje)` przy wywołaniu.

Sprawdzanie `$?` zamiast bezpośredniego wyniku funkcji.

```
sprawdz_dysk
echo "dodatkowy log"
if [ $? -eq 0 ]; then
    echo "dysk OK"
fi
```

Dokładnie ta sama pułapka co przy zwykłych komendach z rozdziału o kodach wyjścia: `$?` po `echo` to kod wyjścia `echo`, nie `sprawdz_dysk`. Naprawa: `if sprawdz_dysk; then` bezpośrednio, bez pośredniego `echo`.

Brak `local` prowadzi do kolizji nazw między funkcjami.

```
i=100

przelicz() {
    for i in 1 2 3; do
        :
    done
}

przelicz
echo "$i"
```

```
3
```

Zewnętrzna zmienna `i` została po cichu nadpisana przez pętlę wewnątrz funkcji. Naprawa: `local i` na początku `przelicz`, albo `for` zapisany jako `for ((i=1; i<=3; i++))` z jawnym `local i` przed pętlą.

□□ Ćwiczenia

> Wszystko poniżej wykonujemy na własnej maszynie, w katalogu `/tmp`. Nic nie wymaga uprawnień administratora.

Ćwiczenie 1 — return kontra echo

```
cd /tmp
jest_parzysta() {
    (( $1 % 2 == 0 ))
}

podwoj() {
    echo $(( $1 * 2 ))
}

for n in 3 4 5; do
    if jest_parzysta "$n"; then
        echo "$n jest parzyste, podwojone: $(podwoj "$n")"
    else
```

```
        echo "$n jest nieparzyste"
    fi
done
```

Które z dwóch funkcji, `jest_parzysta` i `podwoj`, komunikuje wynik przez kod wyjścia, a która przez standardowe wyjście?

Odpowiedź

`jest_parzysta` komunikuje wynik wyłącznie przez kod wyjścia wyrażenia `(( ))` — nie wypisuje niczego, więc jedynym sposobem odczytania jej wyniku jest użycie jej bezpośrednio jako warunku `if`. `podwoj` komunikuje wynik przez `echo` na standardowe wyjście, więc wywołujący musi go złapać przez `$(podwoj ...)` — jej kod wyjścia (zawsze 0, jeśli `echo` się powiedzie) nie niesie żadnej użytecznej informacji.

Ćwiczenie 2 — biblioteka funkcji

```
mkdir -p /tmp/cw_funkcje && cd /tmp/cw_funkcje
cat > lib.sh <<'EOF'
die() {
    echo "BŁĄD: $1" >&2
    exit "${2:-1}"
}
EOF
cat > uzyj.sh <<'EOF'
#!/bin/bash
source "$(dirname "$0")/lib.sh"
[[ -f /brak/takiego/pliku ]] || die "plik konfiguracyjny nie istnieje" 3
echo "nigdy tu nie dotrzemy"
EOF
chmod +x uzyj.sh
./uzyj.sh
echo "kod wyjścia: $?"
```

Odpowiedź

```
BŁĄD: plik konfiguracyjny nie istnieje
kod wyjścia: 3
```

`die` wywołana wewnątrz `uzyj.sh` kończy cały skrypt przez `exit`, nie tylko samą funkcję — dokładnie tak, jak powinna działać funkcja przeznaczona do natychmiastowego przerywania wykonania po błędzie krytycznym.

Pytania kontrolne

1. Dlaczego w funkcji piszemy `return`, a nie `exit`, mimo że oba „kończą coś" z kodem liczbowym?

Odpowiedź

`return` kończy wyłącznie bieżące wywołanie funkcji i oddaje sterowanie do miejsca, z którego ją wywołano — reszta skryptu leci dalej. `exit` kończy cały proces powłoki: skrypt uruchomiony przez `./` albo `bash` przerywa się całkowicie, a skrypt wczytany przez `source` zamyka macierzystą powłokę interaktywną. Funkcja, która ma sygnalizować lokalną porażkę, ale pozwolić reszcie skryptu zdecydować, co dalej, używa `return`; `exit` zostawiamy dla `main` albo dla `die`.

2. Co się stanie, jeśli w funkcji napiszemy `local wynik=$(polecenie_ktore_zawodzi)` pod `set -e`, a `polecenie_ktore_zawodzi` zwróci kod niezerowy?

Odpowiedź

Skrypt się nie zatrzyma. `local` jest tu komendą samą w sobie i to jej kod wyjścia (niemal zawsze 0) widzi `set -e`, a nie kod wyjścia podstawienia komendy po prawej stronie. To jedna z klasycznych, udokumentowanych pułapek `set -e` — naprawiamy ją, rozdzielając deklarację (`local wynik`) i przypisanie (`wynik=$(polecenie_ktore_zawodzi)`) na dwie osobne linie, tak żeby kod wyjścia przypisania nie był zasłonięty przez `local`.

Powiązane: Zmienne, parametry i argumenty · Warunki i pętle · Tryb ścisły: set -euo pipefail · Szkielet skryptu produkcyjnego

Źródła: `man 1 bash` (sekcje *Shell Functions*, *SHELL BUILTIN COMMANDS* → `return`) · `help local` · `help return` · POSIX.1-2024

Revision #11
Created 2026-09-25 00:40:48 UTC by Tomasz Konieczny
Updated 2026-10-03 19:10:18 UTC by Tomasz Konieczny