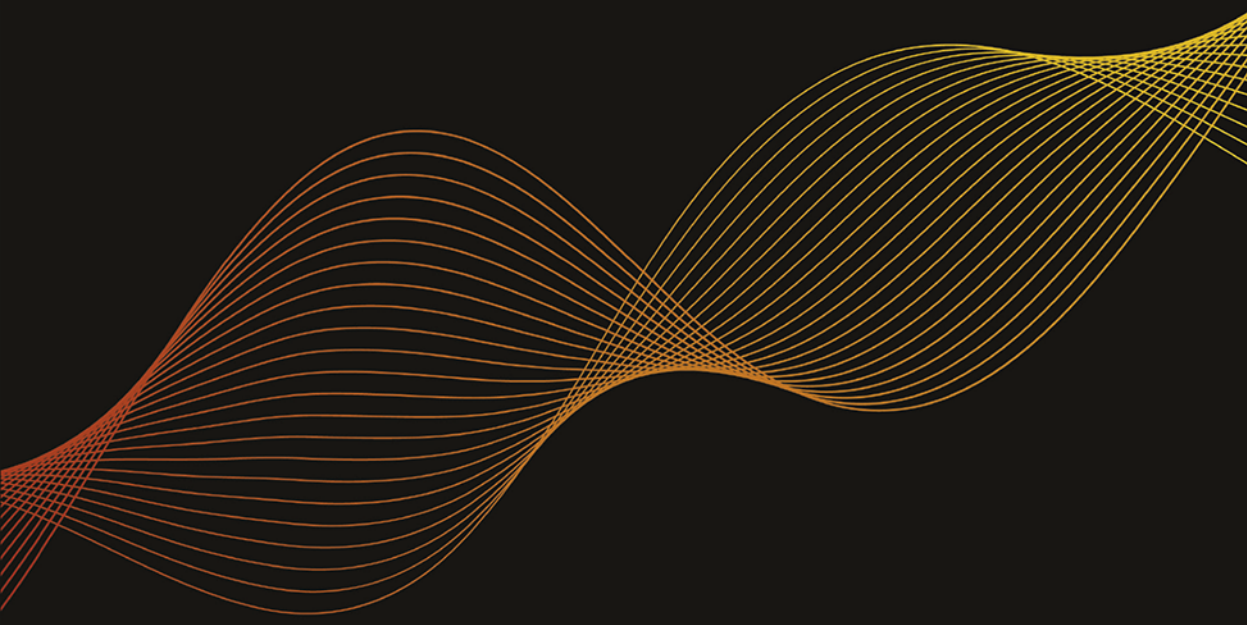
OKIEM EKSPERTA

Linux

Zostań mistrzem skryptów powłoki

Najlepszy przewodnik, z którym zoptymalizujesz, zautomatyzujesz i usprawnisz każde zadanie



Donald A. Tevault

Helion

packt

Tytuł oryginału: The Ultimate Linux Shell Scripting Guide: Automate, Optimize, and Empower tasks with Linux Shell Scripting

Tłumaczenie: Robert Górczyński

ISBN: 978-83-289-3175-6

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
helion.pl/user/opinie/lizomi
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

Rozdział 8

Podstawowa konstrukcja skryptu powłoki

Tak, wiem. Pewnie nie możesz się doczekać, aby rozpocząć tworzenie skryptów powłoki, ale jeszcze nie miałeś okazji tego zrobić. W tym rozdziale omówię podstawy tego zagadnienia. Na koniec zaprezentuję kilka praktycznych i użytecznych przykładowych skryptów.

Wprawdzie wiele technik przedstawionych w tym rozdziale działa w każdej powłoce, ale niektóre mogą być przeznaczone wyłącznie dla powłoki bash. I to na niej się teraz dla uproszczenia skoncentruję. Z kolei w rozdziale 22. pokażę techniki stosowane w powłoce zsh. Natomiast w rozdziale 19. przedstawię metody, które działają w różnych powłokach.

W tym rozdziale omówię następujące zagadnienia:

- podstawy konstrukcji skryptów powłoki,
- wykonywanie testów,
- podpowłoki,
- zmienne w skryptach,
- tablice,
- rozwijanie zmiennych,
- podstawianie poleceń,
- konstrukcje warunkowe i pętle,
- parametry pozycyjne,
- kody wyjścia,
- więcej informacji o poleceniu `echo`,
- przykłady z życia wzięte.

Jeśli jesteś gotowy, zaczynajmy!

Wymagania techniczne

Użyj dowolnej dystrybucji Linuksa z zainstalowaną powłoką bash. Jeśli korzystasz z systemu macOS, użyj jednej z dostępnych maszyn wirtualnych z Linuksem, ponieważ część skryptów wymaga poleceń, które nie będą działać w macOS-ie. Wprawdzie omówione w tekście przykłady możesz wykonywać w swoim lokalnym systemie macOS, ale pamiętaj, że zaprezentuję też ćwiczenia praktyczne.

Dołączyłem również jedno ćwiczenie praktyczne, które wykorzystuje maszynę wirtualną z dystrybucją FreeBSD. Zatem utwórz maszynę wirtualną z tym systemem, a następnie zainstaluj w nim pakiety `sudo` oraz `bash`, jak pokazałem we „Wprowadzeniu" do książki.

Ponadto, jak również wyjaśniłem we „Wprowadzeniu", gotowe skrypty możesz pobrać z repozytorium, które zamieściłem w serwisie GitHub. W tym celu wystarczy wydać następujące polecenie:

```
$ git clone https://github.com/PacktPublishing/The-Ultimate-Linux-Shell-Scripting-
↪Guide.git
```

Podstawy konstrukcji skryptów powłoki

Podczas tworzenia skryptu powłoki pierwszym zadaniem jest określenie powłoki, za pomocą której będzie interpretowany kod danego skryptu.

Uwaga

Możesz mieć konkretny powód, dla którego wybierzesz daną powłokę zamiast innej. Dokładniejsze omówienie tego zagadnienia znajdziesz w rozdziałach 19. i 22.

W pierwszym wierszu skryptu zdefiniujesz powłokę, która ma być użyta jako interpreter. Jest to tzw. **wiersz shebang**. Będzie miał postać podobną do tutaj pokazanej:

```
#!/bin/bash
```

Zazwyczaj wiersz, który rozpoczyna się od znaku `#`, oznacza komentarz i jest ignorowany przez powłokę. Wiersz shebang (i proszę nie pytać, skąd wzięła się ta nazwa) jest wyjątkiem od tej reguły. Oprócz określenia konkretnej powłoki przeznaczonej do użycia, takiej jak `/bin/bash` czy `/bin/zsh`, można też zdefiniować ogólną `/bin/sh`, aby tworzone skrypty były bardziej przenośne i działały w większej liczbie powłok i systemów operacyjnych. Oto jak to wygląda wiersz shebang, który wskazuje ogólną powłokę `sh`:

```
#!/bin/sh
```

Ta ogólna powłoka `sh` ma na celu umożliwienie uruchamiania skryptów w różnych systemach, które mogą (ale nie muszą) mieć zainstalowaną powłokę `bash`. Jednak takie podejście prowadzi również do problemów, ponieważ różne powłoki reprezentowane przez `sh` nie są w pełni ze sobą zgodne. Zobacz, jak to działa w praktyce:

- W systemie FreeBSD oraz prawdopodobnie w innych systemach typu BSD (**Berkeley Software Distribution**) plik wykonywalny `sh` to klasyczna powłoka Bourne'a, która jest przodkiem powłoki bash (Bourne Again Shell).
- W systemach z rodziny Red Hat, `sh` jest dowiązaniem symbolicznym, które wskazuje plik wykonywalny *bash*. Warto pamiętać, że powłoka bash może korzystać z pewnych funkcji programistycznych niedostępnych w innych powłokach, które wymieniłem na tej liście. (Więcej informacji na ten temat przedstawię w rozdziale 19., poświęconym przenośności skryptów powłoki).

- W dystrybucjach typu Debian/Ubuntu `sh` jest dowiązaniem symbolicznym do pliku wykonywalnego *dash*. Ta nazwa oznacza **Debian Almquist Shell**, czyli szybszą i lżejszą implementację powłoki bash.
- W systemie Alpine Linux `sh` wskazuje na `ash`, czyli lekką powłokę, która jest częścią programu `busybox`. (W systemie Alpine powłoka bash nie jest domyślnie zainstalowana).
- W systemie OpenIndiana, który jest **bezpłatnie dostępną wersją** systemu operacyjnego Oracle Solaris, `sh` jest dowiązaniem symbolicznym do powłoki `ksh93`. Znana również jako Korn shell, jest ona w pewnym stopniu, choć nie całkowicie, zgodna z powłoką bash. (Powłoka ksh została opracowana przez Davida Korna).
- W systemie macOS `sh` jest dowiązaniem symbolicznym do powłoki bash. (Co ciekawe, wprawdzie zsh jest domyślną powłoką logowania w macOS-ie, ale powłoka bash jest nadal instalowana domyślnie i pozostaje dostępna do użycia).

Uwaga

Pamiętaj, że używanie w skryptach wiersza shebang w postaci `#!/bin/sh` może być problematyczne. Wynika to z faktu, że reprezentowane przez `#!/bin/sh` poszczególne powłoki w różnych systemach operacyjnych nie są ze sobą w pełni zgodne. Załóżmy, że tworzysz skrypt w systemie z rodziny Red Hat, w którym to `sh` wskazuje powłokę bash. Istnieje duże ryzyko, że ten skrypt nie zadziała w systemach Debian i FreeBSD, w których `sh` wskazuje odpowiednio dash i powłokę Bourne'a. Z tego powodu skoncentruję się na razie na powłoce bash, zaś w prezentowanych przykładach będę używać wiersza shebang w postaci `#!/bin/bash`. Jak wspomniałem już wcześniej, dokładniejsze omówienie związanych z tym kwestii znajdzie się w rozdziale 19., poświęconym przenośności skryptów powłoki.

W zależności od Twoich potrzeb i wymagań skrypt powłoki może być bardzo prosty lub złożony. Może składać się z jednego standardowego polecenia lub kilku takich poleceń systemu Linux/Unix, które będą wykonywane po kolei. Z drugiej strony, skrypty mogą osiągać złożoność zbliżoną do programów napisanych w językach wyższego poziomu, takich jak C.

Na początek spójrz na bardzo prosty skrypt, który składa się z tylko jednego polecenia.

```
#!/bin/bash
rsync -avhe ssh /var/www/html/course/images/
root@192.168.0.22:/var/www/html/course/images/
```

To prosty jednowierszowy skrypt powłoki, którego używałem do tworzenia kopii zapasowej katalogu *images* w jednym komputerze i przesyłania jej do katalogu kopii zapasowych w innym komputerze, działającym pod kontrolą systemu Debian. Ten skrypt wykorzystuje program `rsync` z odpowiednimi opcjami do synchronizacji obu katalogów poprzez sesję bezpiecznej powłoki (ang. *secure shell*, SSH). (Chociaż zazwyczaj nie pozwalam użytkownikowi `root` na logowanie przez `ssh`, w tym przypadku jest to konieczne. Oczywiście takie rozwiązanie stosowałbym tylko w sieci lokalnej i nigdy przez internet). Zgodnie z przeznaczeniem ten skrypt nosi nazwę *rsync_with_debian*. Zanim

będzie można go uruchomić, najpierw trzeba nadać mu uprawnienia do wykonywania. Służy do tego następujące polecenie:

```
$ chmod u+x rsync_with_debian
```

W drugim wierszu omawianego skryptu *rsync_with_debian*, zaraz po wierszu shebang, znajduje się dokładnie to samo polecenie, które musiałbym wpisać w powłoce, gdybym nie miał tego skryptu. Jak widać, dzięki utworzeniu takiego skryptu znacznie uprościłem sobie pracę.

Aby udostępnić ten skrypt wszystkim użytkownikom systemu, umieść go w katalogu */usr/local/bin*, który powinien znajdować się w zmiennej `PATH` każdego użytkownika.

Zanim przejdziesz dalej, utrwal sobie to, czego się właśnie nauczyłeś. W tym celu wykonaj zamieszczone tutaj ćwiczenie praktyczne.

Ćwiczenie praktyczne — zliczanie zalogowanych użytkowników

To ćwiczenie pomoże utworzyć skrypt powłoki, który pokaże liczbę zalogowanych użytkowników. Następnie zmodyfikujesz go w taki sposób, aby wyświetlał tylko unikalnych użytkowników. (W skrypcie wykorzystasz wybrane z filtrów strumieni tekstowych, o których dowiedziałeś się w poprzednich dwóch rozdziałach).

1. W dowolnej z maszyn wirtualnych, które działają pod kontrolą systemu Linux, utwórz trzy dodatkowe konta użytkowników. W systemie Fedora możesz użyć wymienionych tutaj poleceń, przy czym wybierz własne nazwy użytkowników:

   ```
   $ sudo useradd vicky
   $ sudo passwd vicky
   ```

   Z kolei w Debianie wydaj następujące polecenie:

   ```
   $ sudo adduser vicky
   ```

2. W lokalnym terminalu maszyny wirtualnej sprawdź jej adres IP za pomocą wymienionego polecenia:

   ```
   $ ip a
   ```

3. W komputerze gospodarza otwórz cztery okna terminala. Korzystając z adresu IP maszyny wirtualnej, w pierwszym oknie zaloguj się do własnego konta użytkownika, zaś w innych oknach zaloguj się do pozostałych kont. Polecenia będą wyglądać mniej więcej tak:

   ```
   $ ssh vicky@192.168.0.9
   ```

4. W oknie terminala, w którym jesteś zalogowany, wyświetl listę wszystkich aktualnie zalogowanych użytkowników:

   ```
   $ who
   ```

   Powinieneś zobaczyć pięciu użytkowników, ponieważ Twoje konto pojawi się raz dla lokalnego logowania w terminalu i raz dla zdalnego logowania poprzez SSH.

5. Utwórz skrypt o nazwie *logged-in.sh* i następującej zawartości:

```
#!/bin/bash
users="$(who | wc -l)"
echo "There are currently $users users logged in."
```

   Używasz w nim koncepcji **podstawienia polecenia**, aby wynik wykonania polecenia `who | wc -l` przypisać do zmiennej skryptowej `users`. (Więcej informacji na temat tej koncepcji przedstawię w dalszej części książki, więc na razie się tym nie przejmuj).

6. Za pomocą następującego polecenia nadaj skryptowi uprawnienia do wykonywania:

```
$ chmod u+x logged-in.sh
```

7. Teraz uruchom skrypt w ten sposób:

```
$ ./logged-in.sh
```

   Wynik jego uruchomienia powinien wyglądać następująco:

```
There are currently 5 users logged in.
```

   Problem polega na tym, że w rzeczywistości mamy tylko czterech użytkowników. Liczba podana przez skrypt jest niepoprawna, ponieważ Twoja nazwa użytkownika jest liczona dwukrotnie. Teraz naprawisz ten błąd.

8. Zmodyfikuj kod skryptu *logged-in.sh* do następującej postaci:

```
#!/bin/bash
users="$(who | wc -l)"
echo "There are currently $users users logged in."
echo
uniqusers="$(who | cut -d" " -f1 | sort | uniq | wc -l)"
echo "There are currently $uniqusers unique users logged in."
```

   Zmienna `uniqusers` jest tworzona przez wszystkie polecenia połączone ze sobą za pomocą potoku i umieszczone w nawiasie. Polecenie `cut` używa spacji jako separatora (`-d" "`) i wyodrębnia pierwsze pole (`-f1`) z wyniku działania polecenia `who`. Te dane wyjściowe są następnie przekazywane do polecenia `sort`, a później do `uniq`, które z kolei przekazuje do polecenia `wc -l` tylko unikalne nazwy użytkowników.

9. Po ponownym uruchomieniu skryptu dane wyjściowe powinny wyglądać następująco:

```
There are currently 5 users logged in.
There are currently 4 unique users logged in.
```

10. Dokonaj ostatniej modyfikacji skryptu, aby wyświetlić nazwy unikalnych zalogowanych użytkowników. Gotowy skrypt będzie przedstawiał się następująco:

```
#!/bin/bash
users="$(who | wc -l)"
echo "There are currently $users users logged in."
echo
uniqusers="$(who | cut -d" " -f1 | sort | uniq | wc -l)"
echo "There are currently $uniqusers unique users logged in."
echo
```

```
listusers="$(who | cut -d" " -f1 | sort | uniq)"
echo "These users are currently logged in: \n$listusers "
```

Raz jeszcze uruchom skrypt. Powinieneś otrzymać dane wyjściowe podobne do tutaj przedstawionych:

```
There are currently 5 users logged in.
There are currently 4 unique users logged in.
These users are currently logged in:
cleopatra
donnie
frank
vicky
```

Gratulacje! Właśnie utworzyłeś swój pierwszy skrypt powłoki. Teraz czas na jego przetestowanie.

Wykonywanie testów

W trakcie tworzenia skryptów często pojawia się potrzeba sprawdzenia określonych warunków przed podjęciem decyzji o dalszym działaniu. Może to być na przykład weryfikacja istnienia konkretnego pliku lub katalogu, sprawdzenie uprawnień do pliku lub katalogu bądź wiele innych operacji. Istnieją trzy główne sposoby przeprowadzania takich testów:

- Użycie słowa kluczowego `test` wraz z warunkiem testowym, połączone z innym poleceniem za pomocą konstrukcji `&&` lub `||`.
- Umieszczenie warunku testowego w nawiasie kwadratowym.
- Zastosowanie konstrukcji `if-then`.

Najpierw przyjrzysz się słowu kluczowemu `test`.

Używanie słowa kluczowego test

W pierwszym przykładzie sprawdzisz, czy określony katalog istnieje, a jeśli nie — utworzysz go. Oto jak to działa w praktyce:

```
[donnie@fedora ~]$ test -d graphics || mkdir graphics
[donnie@fedora ~]$ ls -ld graphics/
drwxr-xr-x. 1 donnie donnie 0 Sep 26 15:41 graphics/
[donnie@fedora ~]$
```

Teraz dodaj ten kod do skryptu *test_graphics.sh*:

```
#!/bin/bash
cd
pwd
test -d graphics || mkdir graphics
cd graphics
pwd
```

Uruchom skrypt i zobacz, jakie otrzymasz dane wyjściowe:

```
[donnie@fedora ~]$ ./test_graphics.sh
/home/donnie
/home/donnie/graphics
[donnie@fedora ~]$
```

Jak zapewne się domyślasz, operator `-d` oznacza katalog (ang. *directory*). Konstrukcja `||` powoduje, że polecenie `mkdir` zostanie wykonane tylko wtedy, gdy podany katalog nie istnieje. Oczywiście jeśli katalog już istnieje, nie zostanie utworzony ponownie. Jest to dobry środek ostrożności, który może zapobiec przypadkowemu nadpisaniu istniejących plików lub katalogów. (Nieco dalej w rozdziale zamieszczę tabelę z większą liczbą operatorów testowych).

Teraz przyjrzysz się drugiej metodzie przeprowadzania testu.

Umieszczanie warunku testowego w nawiasie kwadratowym

Drugi sposób przeprowadzenia testu polega na umieszczeniu warunku testowego w nawiasie kwadratowym, jak pokazuję w kolejnym przykładzie:

```
[ -d graphics ]
```

Przede wszystkim zwróć uwagę na konieczność umieszczenia spacji po otwierającym nawiasie kwadratowym i przed zamykającym. Przedstawiona konstrukcja sprawdza istnienie katalogu *graphics*, podobnie jak w przypadku polecenia `test -d`. Teraz omawianą konstrukcję umieść w skrypcie *test_graphics_2.sh* w następujący sposób:

```
#!/bin/bash
cd
pwd
[ -d graphics ] || mkdir graphics
cd graphics
pwd
```

Uruchomienie tego skryptu spowoduje wygenerowanie dokładnie takiego samego wyniku jak w przypadku poprzedniego skryptu. Teraz wprowadzisz pewną modyfikację. Kod skryptu *test_graphics_2.sh* zmień w taki sposób, aby przedstawiał się następująco:

```
#!/bin/bash
cd
pwd
[ ! -d graphics ] && mkdir graphics
cd graphics
pwd
```

Użyty tutaj operator negacji (`!`) oznacza, że dany operator wykonuje działanie przeciwne do zamierzonego. W tym przypadku `!` powoduje, że operator `-d` sprawdza brak katalogu *graphics* zamiast jego obecności. Aby takie rozwiązanie zadziałało poprawnie, musisz również zmienić operator `||` na `&&`. (Zwróć uwagę na konieczność umieszczenia spacji operatorem `!` i przed opcją `-d`).

Istnieje możliwość przetestowania wartości liczbowych, na przykład w ten sposób:

```
[ $var -eq 0 ]
```

W tym poleceniu znajduje się odwołanie do wartości zmiennej `var`. Polecenie sprawdza, czy jest ona równa (`-eq`) `0`. Zamiast używać negacji (`!`) do sprawdzenia, czy wartość zmiennej *nie* wynosi `0`, użyj operatora `-ne`. Zobacz, jak to będzie wyglądać w skrypcie *test_var.sh*:

```
#!/bin/bash
var1=0
var2=1
[ $var1 -eq 0 ] && echo "$var1 is equal to zero."
[ $var2 -ne 0 ] && echo "$var2 is not equal to zero."
```

Teraz uruchom wymieniony skrypt:

```
[donnie@fedora ~]$ ./test_var.sh
0 is equal to zero.
1 is not equal to zero.
[donnie@fedora ~]$
```

Trzeci sposób przeprowadzenia testu polega na użyciu konstrukcji `if-then`, na temat której więcej dowiesz się w kolejnym punkcie.

Używanie konstrukcji warunkowej if-then

Konstrukcja `if-then` okazuje się przydatna, gdy masz do czynienia z bardziej złożonymi warunkami. Oto najprostszy przykład zdefiniowany w skrypcie *test_graphics_3.sh*:

```
#!/bin/bash
cd
pwd
if [ ! -d graphics ]; then
        mkdir graphics
fi
cd graphics
pwd
```

Omawiana konstrukcja rozpoczyna się od polecenia `if`, a kończy poleceniem `fi` (którego nazwa to odwrócony zapis słowa kluczowego `if`). Po warunku testowym należy umieścić średnik, a następnie słowo kluczowe `then`. Następny element to akcja, którą chcesz wykonać — w tym przypadku będzie nią polecenie `mkdir graphics`. Wprawdzie w przeciwieństwie do innych języków programowania wcięcie bloku akcji w skryptach powłoki nie jest konieczne, ale pomaga zapewnić większą czytelność skryptu.

Oczywiście możliwości konstrukcji `if-then` są znacznie szersze, niż tutaj przedstawiłem. Jednak teraz nie myśl o tym, ponieważ więcej informacji na ich temat znajdziesz w dalszej części rozdziału. Jednak zanim tam dotrzesz, chcę omówić kilka dodatkowych koncepcji, które możesz stosować, aby urozmaicić tworzone konstrukcje `if-then`.

W następnym punkcie przyjrzysz się różnym rodzajom testów.

Używanie innych rodzajów testów

Istnieje znacznie więcej rodzajów testów, które można przeprowadzić, m.in.: porównanie ciągów tekstowych, porównania liczbowe, sprawdzanie istnienia plików lub katalogów oraz ich uprawnień. W tabeli 8.1 znajduje się zestawienie najpopularniejszych testów wraz z ich operatorami.

Tabela 8.1. Zestawienie najpopularniejszych testów wraz z ich operatorami

Operator	Opis
`-b` *nazwa_pliku*	Prawda, jeśli istnieje plik urządzenia blokowego o podanej nazwie.
`-c` *nazwa_pliku*	Prawda, jeśli istnieje plik urządzenia znakowego o podanej nazwie.
`-d` *nazwa_katalogu*	Prawda, jeśli istnieje katalog o podanej nazwie.
`-e` *nazwa_pliku*	Prawda, jeśli istnieje jakikolwiek plik o podanej nazwie.
`-f` *nazwa_pliku*	Prawda, jeśli istnieje zwykły plik o podanej nazwie.
`-g` *nazwa_pliku*	Prawda, jeśli plik lub katalog mają zdefiniowane uprawnienie SGID.
`-G` *nazwa_pliku*	Prawda, jeśli plik istnieje i należy do efektywnego identyfikatora grupy.
`-h` *nazwa_pliku*	Prawda, jeśli plik istnieje i jest dowiązaniem symbolicznym.
`-k` *nazwa_pliku*	Prawda, jeśli plik lub katalog istnieją i mają ustawiony tzw. bit lepkości (ang. *sticky bit*).
`-L` *nazwa_pliku*	Prawda, jeśli plik istnieje i jest dowiązaniem symbolicznym. (Tak samo jak `-h`).
`-p` *nazwa_pliku*	Prawda, jeśli plik istnieje i jest nazwanym potokiem.
`-O` *nazwa_pliku*	Prawda, jeśli plik istnieje i należy do efektywnego identyfikatora użytkownika.
`-r` *nazwa_pliku*	Prawda, jeśli plik istnieje i jest możliwy do odczytu.
`-S` *nazwa_pliku*	Prawda, jeśli plik istnieje i jest gniazdem.
`-s` *nazwa_pliku*	Prawda, jeśli plik istnieje i ma niezerowy rozmiar.
`-u` *nazwa_pliku*	Prawda, jeśli plik istnieje i ma ustawiony bit SUID.
`-w` *nazwa_pliku*	Prawda, jeśli plik istnieje i jest możliwy do zapisu.
`-x` *nazwa_pliku*	Prawda, jeśli plik istnieje i jest wykonywalny.
plik1 `-nt` *plik2*	Prawda, jeśli *plik1* jest nowszy niż *plik2*.
plik1 `-ot` *plik2*	Prawda, jeśli *plik1* jest starszy niż *plik2*.
`-z` *ciąg_znaków*	Prawda, jeśli długość ciągu znaków wynosi 0.
`-n` *ciąg_znaków*	Prawda, jeśli długość ciągu znaków jest różna od 0.
ciąg1 `==` *ciąg2*	Prawda, jeśli dwa ciągi znaków są identyczne.

Tabela 8.1. Zestawienie najpopularniejszych testów wraz z ich operatorami – ciąg dalszy

Operator	Opis
ciąg1 != *ciąg2*	Prawda, jeśli dwa ciągi znaków nie są identyczne.
ciąg1 < *ciąg2*	Prawda, jeśli *ciąg1* występuje przed *ciąg2* w porządku alfabetycznym.
ciąg1 > *ciąg2*	Prawda, jeśli *ciąg1* występuje po *ciąg2* w porządku alfabetycznym.
liczba1 -eq *liczba2*	Prawda, jeśli dwie liczby całkowite są równe.
liczba1 -ne *liczba2*	Prawda, jeśli dwie liczby całkowite nie są równe.
liczba1 -lt *liczba2*	Prawda, jeśli *liczba1* jest mniejsza od *liczba2*.
liczba1 -gt *liczba2*	Prawda, jeśli *liczba1* jest większa od *liczba2*.
liczba1 -le *liczba2*	Prawda, jeśli *liczba1* jest mniejsza od *liczba2* lub jej równa.
liczba1 -ge *liczba2*	Prawda, jeśli *liczba1* jest większa od *liczba2* lub jej równa.
-o *nazwa_opcji*	Prawda, jeśli opcja powłoki jest włączona.

Wiem, że w tabeli 8.1 znajduje się całkiem sporo informacji. Nie przejmuj się tym. Jeśli nie chcesz tego wszystkiego zapamiętywać, po prostu trzymaj tę tabelę pod ręką, aby móc łatwo po nią sięgnąć.

W następnym podrozdziale przedstawię podpowłokę.

Podpowłoka

Gdy wykonujesz test przy użyciu konstrukcji `[ $var -ne 0 ]`, uruchamia ona tzw. **podpowłokę**. Aby temu zapobiec, użyj zamiast tego następującej konstrukcji:

```
[[ $var -ne 0 ]]
```

Może to sprawić, że skrypt będzie działał nieco wydajniej, co w zależności od konkretnego przypadku będzie miało większe lub mniejsze znaczenie.

Uwaga

Tego typu konstrukcja, `[[. . .]]`, jest również niezbędna podczas wykonywania testów, które wymagają dopasowania wzorca do wyrażenia regularnego. (Dopasowywanie wyrażeń regularnych nie działa w konstrukcji `[. . .]`).

Wadą konstrukcji `[[. . .]]` jest to, że nie można jej używać w niektórych powłokach innych niż bash, na przykład dash, ash czy Bourne'a. (Więcej informacji na ten temat przedstawię w rozdziale 19., poświęconym przenośności skryptów powłoki).

Oczywiście na tym etapie poznawania powłoki nie musisz jeszcze wiedzieć, czym są wyrażenia regularne. Dokładnie o nich opowiem w rozdziale 9., który dotyczy filtrowania tekstu za pomocą narzędzi `grep`, `sed` i właśnie wyrażeń regularnych.

W każdym razie zawsze możesz wypróbować swoje skrypty zarówno z użyciem podpowłok, jak i bez nich, aby sprawdzić, które rozwiązanie sprawdza się lepiej w danej sytuacji.

Omówiłem już wszystkie trzy metody testowania. Nadszedł więc czas na trochę eksperymentów w ramach ćwiczenia praktycznego.

Ćwiczenie praktyczne — testowanie warunków

Do tego ćwiczenia pobierz skrypt *tests-test.sh* z repozytorium GitHuba. (Jest to dość długi skrypt, którego kodu źródłowego nie mogę tutaj zamieścić ze względu na ograniczenia dotyczące formatowania książki). Otwórz skrypt w ulubionym edytorze tekstu i przyjrzyj się strukturze. Zwróć uwagę na operację, która sprawdza istnienie pliku *myfile.txt*. Odpowiedni fragment kodu wygląda mniej więcej tak:

```
#!/bin/bash
[ -f myfile.txt ] && echo "This file exists." || echo "This file does not exist."
```

Następnie zobaczysz polecenie utworzenia pliku, jeśli jeszcze nie istnieje:

```
echo "We will now create myfile.txt if it does not exist, and make it with only
read permissions for $USER."
[ -f myfile.txt ] || touch myfile.txt
```

Dalej znajduje się polecenie, które plikowi *myfile.txt* nadaje uprawnienia w postaci `400`. Zgodnie z nimi użytkownik ma możliwość odczytywania pliku, ale nikt nie ma prawa do jego zapisu. Potem chcesz zweryfikować, czy wszystkie uprawnienia do zapisu zostały usunięte. Oto jak przedstawia się odpowiedzialny za to fragment kodu:

```
chmod 400 myfile.txt
ls -l myfile.txt
echo
echo "We will now see if myfile.txt is writable."
[ -w myfile.txt ] && echo "This file is writable." || echo "This file is not
writable."
```

Po kilku kolejnych modyfikacjach ustawień uprawnień i testach z nimi związanych zobaczysz fragment kodu, który sprawdza istnienie katalogu i tworzy go, jeśli jeszcze nie istnieje:

```
[ -d somedir ] || echo "somedir does not exist."
[ -d somedir ] || mkdir somedir && echo "somedir has just been created."
ls -ld somedir
```

Pod koniec skryptu zobaczysz fragment kodu, który testuje stan opcji `noclobber`, włącza ją, a następnie ponownie sprawdza jej stan.

Po przejrzeniu skryptu uruchom go, aby zobaczyć, jaki będzie wynik jego działania.

Zadanie dla ambitnych: przepisz ten skrypt do własnego pliku. Dlaczego? Cóż, mały sekret polega na tym, że samodzielne wpisywanie kodu pomaga lepiej zrozumieć przedstawione w nim koncepcje.

W następnym kroku utwórz skrypt o nazwie *tests-test_2.sh* i następującej zawartości:

```
#!/bin/bash
echo "We will now compare text strings."
string1="abcd"
string2="efgh"
[[ $string1 > $string2 ]] && echo "string1 comes after string2 alphabetically." ||
↪echo "string1 comes before string2 alphabetically."
echo
echo "We will now compare numbers."
num1=10
num2=9
[[ $num1 -gt $num2 ]] && echo "num1 is greater than num2." || echo "num1 is less
↪than num2."
```

Nadaj skryptowi uprawnienia do wykonywania, a następnie uruchom go, aby zobaczyć wygenerowane dane wyjściowe. Zmień wartości zmiennych `string1`, `string2`, `num1` i `num2`, po czym ponownie uruchom skrypt i przeanalizuj otrzymany wynik.

Koniec ćwiczenia.

W następnym podrozdziale bliżej przyjrzysz się zmiennym.

Zmienne w skryptach

Już trochę opowiedziałem o zmiennych w skryptach i pokazałem przykłady ich użycia. Ale to nie wszystko, jest jeszcze kilka aspektów, o których warto wspomnieć.

Tworzenie i usuwanie zmiennych

Jak wcześniej wyjaśniłem, czasami konieczne lub po prostu wygodniejsze jest definiowanie zmiennych w skryptach. Możesz także definiować, przeglądać i usuwać zmienne bezpośrednio z poziomu powłoki. Oto przykład pracy ze zmiennymi:

```
[donnie@fedora ~]$ car=Ford
[donnie@fedora ~]$ echo $car
Ford
[donnie@fedora ~]$ unset car
[donnie@fedora ~]$ echo $car
[donnie@fedora ~]$
```

W tym przykładzie zdefiniowałeś zmienną `car` i przypisałeś jej wartość `Ford`. Pierwsze polecenie `echo` wyświetla aktualną wartość zmiennej. Natomiast drugie potwierdza pomyślne usunięcie zmiennej za pomocą polecenia `unset`.

Zmienne i poziomy powłoki

Gdy umieszczasz wiersz shebang, taki jak `#!/bin/bash` lub `#!/bin/sh`, na początku skryptu, wówczas nowa nieinteraktywna powłoka potomna zostanie uruchomiona za każdym razem, gdy uruchomisz ten skrypt. Po zakończeniu jego działania także uruchomiona przez niego powłoka kończy działanie. Powłoka potomna dziedziczy wszystkie

zmienne, które zostały wyeksportowane z powłoki nadrzędnej. Jednak powłoka nadrzędna nie dziedziczy żadnych zmiennych po powłoce potomnej. Przekonasz się o tym, gdy w powłoce nadrzędnej przypiszesz zmiennej `car` wartość `Volkswagen` w następujący sposób:

```
[donnie@fedora ~]$ export car="Volkswagen"
[donnie@fedora ~]$ echo $car
Volkswagen
[donnie@fedora ~]$
```

Następnie utwórz skrypt o nazwie *car_demo.sh* i następującej zawartości:

```
#!/bin/bash
echo \$car is set to $car
export car=Toyota
echo "The $car is very fast."
echo \$car is set to $car
```

Nowemu plikowi nadaj uprawnienia do wykonywania, a następnie uruchom skrypt. Otrzymasz pokazane tutaj dane wyjściowe:

```
[donnie@fedora ~]$ ./car_demo.sh
$car is set to Volkswagen
The Toyota is very fast.
$car is set to Toyota
[donnie@fedora ~]$
```

Zwróć uwagę, jak wartość zmiennej `car` (tutaj `Volkswagen`) została dziedziczona po powłoce nadrzędnej. Dzieje się tak z powodu użycia polecenia `export` — gwarantuje ono, że wartość zmiennej będzie dostępna dla powłoki potomnej, w której został uruchomiony skrypt. Spróbuj raz jeszcze, ale tym razem bez eksportowania zmiennej:

```
[donnie@fedora ~]$ unset car
[donnie@fedora ~]$ car=Studebaker
[donnie@fedora ~]$ ./car_demo.sh
$car is set to
The Toyota is very fast.
$car is set to Toyota
[donnie@fedora ~]$
```

Aby rozwiązanie zadziałało, najpierw trzeba usunąć zmienną `car` za pomocą polecenia `unset`. Po użyciu tego polecenia wartość zmiennej nie będzie dłużej eksportowana. Gdy teraz uruchomisz skrypt, nie znajdzie on wartości zmiennej `car`, zdefiniowanej w powłoce nadrzędnej.

Dlaczego to jest ważne? Chodzi o to, że czasami będziesz tworzyć skrypty, które z kolei będą wywoływać inne skrypty, co w efekcie prowadzi do uruchomienia kolejnej powłoki potomnej. Jeśli chcesz, aby zmienne były dostępne dla powłoki potomnej, musisz je wyeksportować.

Rozróżnianie wielkości liter

W nazwach zmiennych rozróżniana jest wielkość liter. Oznacza to, że zmienna o nazwie `car` jest zupełnie inna niż zmienne o nazwach `Car` czy `CAR`.

Zmienne środowiskowe mają nazwy zapisane wyłącznie wielkimi literami. Dobrą praktyką jest natomiast używanie małych liter lub różnej wielkości liter w nazwach zmiennych programistycznych. Co ciekawe, powłoki Linux i Unix nie wymuszają tej reguły dla zmiennych programistycznych. Warto jednak jej przestrzegać, ponieważ pomaga to uniknąć przypadkowego nadpisania wartości ważnej zmiennej środowiskowej.

Uwaga

Z przykrością muszę stwierdzić, że w internecie można znaleźć poradniki dotyczące skryptów powłoki, w których autorzy zachęcają do tworzenia nazw zmiennych wyłącznie wielkimi literami. Niedawno natrafiłem na jeden z takich poradników. Wprawdzie w większości przypadków autor proponuje tworzenie zmiennych o nazwach, które nie kolidują ze zmiennymi środowiskowymi, ale w jednym nakazuje utworzyć zmienną `USER` i przypisać jej pewną wartość. Nie muszę dodawać, że `USER` to nazwa już istniejącej zmiennej środowiskowej. Zatem przypisanie jej nowej wartości spowoduje nadpisanie tej, która powinna tam być. Wniosek z tego taki, że w internecie można znaleźć wiele dobrych poradników, ale niestety jest też sporo takich, które przekazują błędne informacje.

Zmienne tylko do odczytu

Jak już wyjaśniłem, po zadeklarowaniu zmiennej w standardowy sposób można ją usunąć lub przypisać jej nową wartość. Istnieje również możliwość zdefiniowania zmiennej jako tylko do odczytu, co uniemożliwia jej ponowne zdefiniowanie lub usunięcie. Oto jak to działa:

```
[donnie@fedora ~]$ car=Nash
[donnie@fedora ~]$ echo $car
Nash
[donnie@fedora ~]$ readonly car
[donnie@fedora ~]$ unset car
bash: unset: car: cannot unset: readonly variable
[donnie@fedora ~]$ car=Hudson
bash: car: readonly variable
[donnie@fedora ~]$
```

Gdy właściwość została zdefiniowana jako przeznaczona tylko do odczytu, jedynym sposobem na zmianę lub usunięcie zmiennej `car` jest zamknięcie okna terminala.

Uwaga

Cóż, jest to jedyny sposób na pozbycie się zmiennej tylko do odczytu, jeśli nie masz uprawnień administratora. Natomiast jeżeli posiadasz takie uprawnienia, możesz użyć **debuggera GNU bash** (gdb) do usunięcia takiej zmiennej. Jednak wyjaśnienie tego zagadnienia wykracza poza zakres tematyczny rozdziału. (Dokładniejsze omówienie narzędzia `gdb` znajdziesz w rozdziale 21., poświęconym debugowaniu skryptów powłoki).

To wszystko sprawdza się świetne, o ile chcesz zdefiniować tylko pojedyncze zmienne. Jednak co w sytuacji, gdy potrzebujesz całej listy zmiennych? W takim przypadku przydają się tablice. Przyjrzysz się im w następnym podrozdziale.

Tablice

Tablica pozwala zebrać listę elementów w jednej zmiennej. Najprostszym sposobem utworzenia tablicy jest przypisanie wartości jednemu z jej indeksów, jak pokazuję w kolejnym przykładzie:

```
nazwa[indeks]=wartość
```

W tym przypadku `nazwa` to nazwa tablicy, zaś `indeks` to położenie elementu w tablicy. (Pamiętaj, że `indeks` musi być liczbą). Natomiast `wartość` to konkretna wartość przypisana danemu elementowi tablicy.

Numeracja elementów w tablicy zaczyna się od `0`. Zatem zapis `nazwa[0]` odwołuje się do pierwszego elementu tablicy. Aby utworzyć tablicę indeksowaną, użyj polecenia `declare` z opcją `-a`:

```
[donnie@fedora ~]$ declare -a myarray
[donnie@fedora ~]$
```

Teraz utwórz listę, która zostanie wstawiona do tablicy:

```
[donnie@fedora ~]$ myarray=(item1 item2 item3)
[donnie@fedora ~]$
```

Wprawdzie możesz sprawdzić wartość dowolnego elementu tablicy, ale istnieje specjalny sposób, aby to zrobić. Oto jak to wygląda:

```
[donnie@fedora ~]$ echo ${myarray[0]}
item1
[donnie@fedora ~]$ echo ${myarray[1]}
item2
[donnie@fedora ~]$ echo ${myarray[2]}
item3
[donnie@fedora ~]$
```

Zwróć uwagę na umieszczenie konstrukcji `myarray[x]` w nawiasie klamrowym.

Aby wyświetlić całą listę elementów tablicy, użyj znaku `*` lub `@` zamiast numeru indeksu, jak pokazuję w kolejnym przykładzie:

```
[donnie@fedora ~]$ echo ${myarray[*]}
item1 item2 item3
[donnie@fedora ~]$ echo ${myarray[@]}
item1 item2 item3
[donnie@fedora ~]$
```

Aby po prostu policzyć liczbę elementów tablicy, wystarczy wstawić znak `#` przed jej nazwą:

```
[donnie@fedora ~]$ echo ${#myarray[@]}
3
[donnie@fedora ~]$ echo ${#myarray[*]}
3
[donnie@fedora ~]$
```

W ten sposób przedstawiłem podstawy pracy z tablicami. Ćwiczenie zamieszczone w następnym punkcie pozwoli zająć się czymś bardziej praktycznym.

Ćwiczenia praktyczne — praca z tablicami

1. Aby zobaczyć, jak tworzy się tablice, rozpocznij od zdefiniowania skryptu *ip.sh* o następującej zawartości:

```
#!/bin/bash
echo "IP Addresses of intruder attempts"
declare -a ip
ip=( 192.168.3.78 192.168.3.4 192.168.3.9 )
echo "ip[0] is ${ip[0]}, the first item in the list."
echo "ip[2] is ${ip[2]}, the third item in the list."
echo "******************************"
echo "The most dangerous intruder is ${ip[1]}, which is in ip[1]."
echo "******************************"
echo "Here is the entire list of IP addresses in the array."
echo ${ip[*]}
```

2. Plikowi nadaj uprawnienia do wykonywania i uruchom go.

```
$ chmod u+x ip.sh
$ ./ip.sh
```

3. Utwórz katalog */opt/scripts/* przeznaczony do przechowywania plików z danymi, które będą potrzebne Twoim skryptom. Możesz to zrobić w następujący sposób:

```
$ sudo mkdir /opt/scripts
```

4. W katalogu */opt/scripts/* utwórz plik o nazwie *banned.txt*. (Pamiętaj, że w tym katalogu będziesz musiał użyć `sudo` w trakcie uruchamiania edytora tekstu). W nowym pliku umieść następującą zawartość:

```
192.168.0.48
24.190.78.101
38.101.148.126
41.206.45.202
58.0.0.0/8
59.107.0.0/17
59.108.0.0/15
59.110.0.0/15
59.151.0.0/17
59.155.0.0/16
59.172.0.0/15
```

5. W katalogu domowym utwórz skrypt o nazwie *attackers.sh*, który zbuduje tablicę zabronionych adresów IP zamieszczonych na liście pochodzącej z utworzonego wcześniej pliku tekstowego. Następnie w skrypcie *attackers.sh* umieść następującą zawartość:

```
#!/bin/bash
badips=$(cat /opt/scripts/banned.txt)
declare -a attackers
attackers=( $badips )
echo "Here is the complete list: "
echo ${attackers[@]}
echo
echo "Let us now count the items in the list."
num_attackers=${#attackers[*]}
```

```
echo "There are $elements IP addresses in the list."
echo
echo "attackers[2] is ${attackers[2]}, which is the third address in the list."
exit
```

6. Skryptowi *attackers.sh* nadaj uprawnienia do wykonywania i uruchom go.

```
$ chmod u+x attackers.sh
$ ./attackers.sh
```

7. Zmodyfikuj skrypt, aby elementy o indeksach `0`, `5` i `8` zostały wyświetlone na ekranie, a następnie ponownie uruchom skrypt. (Już wcześniej wyjaśniłem, jak to zrobić).

Koniec ćwiczenia.

Kilka słów wyjaśnienia na temat tego skryptu. Przede wszystkim w drugim wierszu zostało użyte podstawienie polecenia `cat`, aby zawartość pliku *banned.txt* przypisać do zmiennej `badips`. (Wiem, że ciągle pokazuję przykłady podstawienia polecenia, ale jeszcze nie wyjaśniłem w pełni tej koncepcji. Nie martw się, ponieważ zrobię to za chwilę). Jednak to wciąż nie jest tablica. Utworzona jest oddzielnie, za pomocą polecenia `declare -a`. Następnie w wierszu `attackers=` znajduje się odwołanie do wartości zmiennej `badips`, która została później użyta do utworzenia tablicy `attackers`. Ewentualnie można pominąć użycie zmiennej pośredniej i utworzyć tablicę bezpośrednio, z wykorzystaniem podstawienia polecenia `cat`, jak możesz to zobaczyć w skrypcie *attackers_2.sh*:

```
#!/bin/bash
declare -a badips
badips=( $(cat /opt/scripts/banned.txt) )
echo "Here is the complete list: "
echo ${badips[@]}
echo
echo "Let us now count the items in the list."
elements=${#badips[*]}
echo "There are $elements IP addresses in the list."
echo
echo "badips[2] is ${badips[2]}, which is the third address in the list."
exit
```

Obie metody działają, ale omówiona powyżej jest nieco bardziej przejrzysta.

Uwaga

W rzeczywistym scenariuszu można jeszcze dodać kod odpowiedzialny za automatyczne utworzenie reguły zapory sieciowej, która będzie blokowała wszystkie adresy wymienione w pliku *banned.txt*. Jednak wówczas jest wymagane zastosowanie technik, których jeszcze nie omówiłem. Dlatego takie rozwiązanie zaprezentuję nieco później.

W następnym podrozdziale zajmiesz się wykorzystaniem zmiennych.

Rozwijanie zmiennych

Rozwijanie zmiennych, nazywane również **rozwijaniem parametrów**, pozwala powłoce testować lub modyfikować wartości zmiennej, która jest używana w skrypcie. Odbywa się to za pomocą specjalnych modyfikatorów ujętych w nawiasy klamrowe i poprzedzonych znakiem dolara (`${zmienna}`). Jeśli dana zmienna nie jest zdefiniowana w powłoce bash, zostanie rozwinięta do pustego ciągu tekstowego. Najlepszym sposobem na zrozumienie sposobu działania tego mechanizmu jest przeanalizowanie kilku prostych przykładów.

Przypisywanie wartości do niezainicjalizowanej zmiennej

Najpierw pokażę przykład zdefiniowania zmiennej `cat` o wartości w postaci imienia mojego 16-letniego szarego kociaka. Następnie wykonasz test, aby sprawdzić, czy zmienna `cat` rzeczywiście ma przypisaną wartość:

```
[donnie@fedora ~]$ cat=Vicky
[donnie@fedora ~]$ echo ${cat-"This cat variable is not set."}
Vicky
[donnie@fedora ~]$
```

Teraz usuń wartość zmiennej `cat` i ponownie wykonaj test. Zobaczmy, jaki będzie efekt:

```
[donnie@fedora ~]$ unset cat
[donnie@fedora ~]$ echo ${cat-"This cat variable is not set."}
This cat variable is not set.
[donnie@fedora ~]$
```

Co się więc stało? Otóż, znak -, który znajduje się między elementami `cat` i `This cat variable is not set`, sprawdza, czy zmienna `cat` ma przypisaną wartość. Jeśli nie ma, wówczas ciąg tekstowy umieszczony po znaku - zostaje użyty w miejsce wartości zmiennej. Jednak ta podstawiona wartość nie jest faktycznie *przypisywana* zmiennej, co widać w kolejnym przykładzie:

```
[donnie@fedora ~]$ echo $cat
[donnie@fedora ~]$
```

Teraz zmiennej `cat` przypisz wartość `null` i spróbuj ponownie:

```
donnie@fedora:~$ cat=
donnie@fedora:~$ echo ${cat-"This cat variable is not set."}
donnie@fedora:~$
```

Tym razem otrzymujesz dane wyjściowe w postaci pustego wiersza, ponieważ zmienna `cat` została zdefiniowana. Jednak przypisaną jej wartością jest `null`. Spróbuj jeszcze raz, przy czym użyj znaków `:-` zamiast - w następujący sposób:

```
donnie@fedora:~$ echo ${cat:-"This cat variable is not set."}
This cat variable is not set.
donnie@fedora:~$
```

Dzieje się tak, ponieważ umieszczenie dwukropka przed znakiem minus powoduje, że zmienne, którym przypisano wartość `null`, są traktowane jak zmienne niezdefiniowane.

Na tym kończę temat zmiennych niezdefiniowanych. Czasami może zachodzić potrzeba pracy ze zmiennymi, które mają już przypisane wartości. Tym zajmiesz się w następnym punkcie.

Przypisywanie wartości do zmiennej zbioru

Możesz też postąpić odwrotnie i podstawić wartość w miejsce zmiennej, która ma już przypisaną wartość. Przykład takiego rozwiązania wygląda następująco:

```
[donnie@fedora ~]$ car="1958 Edsel Corsair"
[donnie@fedora ~]$ echo ${car+"car is set and might or might not be null"}
car is set and might or might not be null
[donnie@fedora ~]$
```

W tym przypadku konstrukcja z użyciem znaku + powoduje, że następujący po nim ciąg tekstowy zostaje podstawiony w miejsce przypisanej wartości zmiennej. Warto zauważyć, że w tym ciągu tekstowym nie ma żadnych znaków specjalnych, które wymagałyby znaków cytowania, więc są one opcjonalne. Niemniej jednak dobrą praktyką jest używanie — dla bezpieczeństwa — znaków cytowania. Należy również pamiętać, że taka operacja podstawienia nie zmienia faktycznie przypisanej wartości zmiennej `car`, co można zaobserwować w kolejnym przykładzie:

```
[donnie@fedora ~]$ echo $car
1958 Edsel Corsair
[donnie@fedora ~]$
```

Jak właśnie pokazałem na przykładzie operatora –, operator + traktuje zmienne o wartości `null` jako zdefiniowane. Natomiast jeśli chcesz takie zmienne uznawać za niezdefiniowane, wówczas użyj operatora `:+`. Gdy utworzysz zmienną i pozostawisz ją z wartością `null`, będzie to wyglądać mniej więcej tak:

```
[donnie@fedora ~]$ computer=
[donnie@fedora ~]$ echo ${computer:+"computer is set and is not null"}
[donnie@fedora ~]$
```

W przypadku wartości `null` polecenie `echo` nie generuje żadnych danych wyjściowych. Akurat korzystam z komputera Dell, więc zmiennej `computer` przypisz teraz wartość `Dell`, jak pokazuję poniżej:

```
[donnie@fedora ~]$ computer=Dell
[donnie@fedora ~]$ echo ${computer:+"computer is set and might or might not be null"}
computer is set and might or might not be null
[donnie@fedora ~]$
```

Jak już wcześniej wspomniałem, omówione właśnie operatory podstawiają wartość zmiennej w zależności od tego, czy ma ona przypisaną wartość. Jednak nie zmienią one faktycznej wartości zmiennej. Czasami może zachodzić potrzeba zmiany wartości zmiennej i taką sytuację omówię w następnym punkcie.

Przypisywanie wartości zmiennej

Kolejna sztuczka pozwala przypisywać wartość niezainicjalizowanej zmiennej przy użyciu operatorów = oraz :=. Zacznij od przypisania wartości zmiennej town:

```
donnie@fedora:~$ unset town
donnie@fedora:~$ echo $town
donnie@fedora:~$ echo ${town="Saint Marys"}
Saint Marys
donnie@fedora:~$ echo $town
Saint Marys
donnie@fedora:~$
```

Teraz zobacz, czy możesz przypisać inną wartość tej zmiennej:

```
donnie@fedora:~$ echo ${town="Kingsland"}
Saint Marys
donnie@fedora:~$ echo $town
Saint Marys
donnie@fedora:~$
```

Jak już wcześniej pokazałem, użycie tego operatora bez poprzedzającego go dwukropka powoduje, że zmienna o wartości null jest traktowana jako zdefiniowana. Spójrz na kolejny przykład:

```
donnie@fedora:~$ unset town
donnie@fedora:~$ town=
donnie@fedora:~$ echo ${town="Saint Marys"}
donnie@fedora:~$
```

Aby zobaczyć, jak działa operator :=, utwórz zmienną armadillo o wartości null, a następnie przypisz jej wartość domyślną w następujący sposób:

```
[donnie@fedora ~]$ armadillo=
[donnie@fedora ~]$ echo ${armadillo:=Artie}
Artie
[donnie@fedora ~]$ echo $armadillo
Artie
[donnie@fedora ~]$
```

Artie to tymczasowe imię, jakie nadałem pancernikowi, który niedawno zaczął w nocy odwiedzać mój ogród. Jednak jeszcze nie wiem, czy ten pancernik to samiec czy samica, więc nie jestem pewien, czy Artie będzie odpowiednim imieniem. Jeśli okaże się, że to samica, być może zmienię imię na Annie. Spróbuj więc powtórzyć poprzednie ćwiczenie, ale z pancernikiem o imieniu Annie. Następnie zobaczysz, czy można użyć przykładowej zmiennej, aby zmienić jej wartość z powrotem na Artie:

```
[donnie@fedora ~]$ armadillo=Annie
[donnie@fedora ~]$ echo ${armadillo:=Artie}
Annie
[donnie@fedora ~]$ echo $armadillo
Annie
[donnie@fedora ~]$
```

Zmienna armadillo miała już przypisaną wartość Annie, dlatego, jak widzisz, polecenie echo ${armadillo:=Artie} nie przyniosło żadnego efektu, poza wyświetleniem wartości, która wcześniej była przypisana zmiennej.

Co w sytuacji, jeśli nie chcesz podstawiać wartości zmiennej, a jedynie wyświetlić komunikat błędu? Warto przyjrzyeć się temu bliżej.

Wyświetlanie komunikatu błędu

Nie zawsze chcesz wykonywać podstawienie wartości lub przypisanie wartości dla niezdefiniowanej zmiennej. Czasami w takiej sytuacji wystarczające będzie po prostu wyświetlenie komunikatu błędu (za pomocą standardowego strumienia błędów). Do tego celu można wykorzystać konstrukcję `:?` w następujący sposób:

```
[donnie@fedora ~]$ dog=
[donnie@fedora ~]$ echo ${dog:?The dog variable is unset or null.}
bash: dog: The dog variable is unset or null.
[donnie@fedora ~]$
```

Spróbuj to ponownie zrobić z psem o imieniu `Rastus`. Tak wabił się owczarek angielski, którego miała moja babcia, gdy byłem dzieckiem. Niezbędne polecenia przedstawiają się następująco:

```
[donnie@fedora ~]$ dog=Rastus
[donnie@fedora ~]$ echo ${dog:?The dog variable is unset or null.}
Rastus
[donnie@fedora ~]$
```

Prawdopodobnie myślisz sobie, że operacja wygląda dokładnie tak samo jak w pierwszym przykładzie, w którym za pomocą znaku - wartość niezdefiniowanej zmiennej `cat` zastąpiłem odpowiednim komunikatem. Cóż, po części masz rację. Różnica polega na tym, że konstrukcja oparta na znaku - podstawia wartość pojawiającą się na standardowym wyjściu, podczas gdy oparta na znakach `:?` podstawia komunikat, który pojawia się na standardowym wyjściu błędów. Ponadto jeśli w skrypcie powłoki użyjesz konstrukcji `:?` w połączeniu z niezdefiniowaną zmienną, spowoduje to zakończenie działania skryptu.

Wypróbuj to samodzielnie. W tym celu utwórz skrypt o nazwie *ex.sh* i następującej zawartości:

```
#!/bin/bash
var=
: ${var:?var is unset, you big dummy}
echo "I wonder if this will work."
```

Do tej pory pokazałem, jak używać polecenia `echo` do podstawiania zmiennych i wyświetlania danych wyjściowych. Ta konkretna konstrukcja poprzez użycie znaku `:` zamiast polecenia `echo` pozwala na samo testowanie zmiennej bez wyświetlania jakichkolwiek wyników. Gdy teraz uruchomisz ten skrypt, zobaczysz, że kończy on działanie przed wykonaniem ostatniego polecenia `echo`:

```
[donnie@fedora ~]$ ./ex.sh
./ex.sh: line 3: var: var is unset, you big dummy
[donnie@fedora ~]$
```

Chwileczkę! Czy ja właśnie nazwałem siebie wielkim głupkiem (ang. *big dummy*)? No cóż. W każdym razie zmień skrypt, aby zmienna miała przypisaną wartość, jak widać na przykładzie skryptu *ex_2.sh*:

```
#!/bin/bash
var=somevalue
: ${var:?"var is unset, you big dummy"}
echo "I wonder if this will work with a value of "$var"."
```

Teraz skrypt wykonuje się do końca, co możesz zobaczyć tutaj:

```
donnie@fedora:~$ ./ex_2.sh
I wonder if this will work with a value of somevalue.
donnie@fedora:~$
```

Podsumowując, użycie dwukropka zamiast polecenia `echo` zapobiega wyświetlaniu wartości zmiennej w konstrukcji `${var:?"var is unset, you big dummy"}`. Możesz zmienić ten sposób działania — wystarczy z powrotem zastąpić dwukropek poleceniem `echo`, jak pokazuję w kodzie pliku *ex_3.sh*:

```
#!/bin/bash
var=somevalue
echo ${var:?"var is unset, you big dummy"}
echo "I wonder if this will work with a value of "$var"."
```

Zobacz, jaki będzie wynik tej zmiany:

```
donnie@fedora:~$ ./ex_3.sh
somevalue
I wonder if this will work with a value of somevalue.
donnie@fedora:~$
```

Tym razem wartość zmiennej `var` została wyświetlona.

Uwaga

Jak właśnie pokazałem na przykładzie operatorów `?` i `+`, poprzedzenie znaku `?` dwukropkiem powoduje, że taki operator traktuje zmienną utworzoną z wartością `null` jako niezdefiniowaną. Natomiast pominięcie dwukropka spowoduje, że tego rodzaju zmienna będzie uznana za zdefiniowaną.

W następnym podrozdziale przejdę do omówienia przesunięć zmiennych.

Używanie przesunięć zmiennych

Ostatni typ rozwijania zmiennych, który zamierzam przedstawić, dotyczy podstawiania tylko fragmentu ciągu tekstowego. Ten mechanizm wykorzystuje **przesunięcie zmiennej** i może być nieco trudniejszy do zrozumienia bez konkretnego przykładu.

Gdy definiujesz zmienną, ma ona określoną wielkość, czyli liczbę znaków. Konstrukcja `${zmienna:przesunięcie}` używa przesunięcia, czyli liczby znaków od określonego miejsca. Jeśli przesunięcie wynosi `4`, konstrukcja pominie pierwsze cztery znaki i wyświetli tylko znaki począwszy od piątego. Dodanie parametru `długość` w konstrukcji `${zmienna:przesunięcie:długość}` pozwala również określić, ile znaków chcesz użyć. Na początek utwórz zmienną tekstową o wartości `MailServer` w następujący sposób:

```
[donnie@fedora ~]$ text=MailServer
[donnie@fedora ~]$ echo $text
MailServer
[donnie@fedora ~]$
```

Załóżmy teraz, że chcesz zobaczyć tylko tekst rozpoczynający się od piątego znaku. Możesz użyć przesunięcia w pokazany tutaj sposób:

```
[donnie@fedora ~]$ echo ${text:4}
Server
[donnie@fedora ~]$
```

Świetnie, to działa. Teraz załóżmy, że chcesz wyświetlić tylko pierwsze cztery litery. Użyj więc przesunięcia i długości w następujący sposób:

```
[donnie@fedora ~]$ echo ${text:0:4}
Mail
[donnie@fedora ~]$
```

Takie wywołanie oznacza, że odczyt danych rozpoczyna się od pozycji następującej po zerowej i obejmuje tylko pierwsze cztery znaki.

Możesz również wyodrębnić fragment tekstu z dowolnego miejsca ciągu tekstowego, jak pokazuję w kolejnym przykładzie:

```
[donnie@fedora ~]$ echo ${text:4:5}
Serve
[donnie@fedora ~]$
```

W tym przykładzie dane są odczytywane począwszy od piątego znaku i obejmują pięć kolejnych.

Przedstawię teraz znacznie praktyczniejsze zastosowania omawianej konstrukcji. Załóżmy, że masz zmienną `location`, która zawiera nazwę miasta i stanu w USA wraz z odpowiadającym mu *kodem pocztowym*. Powiedzmy, że z podanego ciągu tekstowego chcesz wyodrębnić sam kod pocztowy. Oto jak możesz to zrobić:

```
[donnie@fedora ~]$ location="Saint Marys GA 31558"
[donnie@fedora ~]$ echo "Zip Code: ${location:14}"
Zip Code:  31558
[donnie@fedora ~]$
```

Zamiast definiować przesunięcie poprzez odliczanie od początku ciągu tekstowego, można również użyć liczby ujemnej i tym samym wyodrębnić tylko ostatni fragment danych. Ponieważ kod pocztowy składa się z pięciu cyfr, można użyć wartości `-5` w następujący sposób:

```
[donnie@fedora ~]$ echo "Zip Code: ${location: -5}"
Zip Code: 31558
[donnie@fedora ~]$
```

Aby zapewnić poprawne działanie tego mechanizmu, zawsze pamiętaj o pozostawieniu spacji między dwukropkiem i myślnikiem. Ponadto ponieważ nazwy miast mogą mieć różną długość, ta metoda będzie lepszym rozwiązaniem, jeśli chcesz wyodrębniać kody pocztowe z całej listy lokalizacji.

I to tyle jeśli chodzi o przesunięcia. Teraz zajmiesz się dopasowywaniem wzorców.

Dopasowanie wzorca

Kolejna sztuczka związana z interpretacją zmiennych dotyczy dopasowywania wzorców. Rozpoczniesz od utworzenia zmiennej `pathname` w przedstawiony tutaj sposób:

```
[donnie@fedora ~]$ pathname="/var/lib/yum"
[donnie@fedora ~]$
```

Załóżmy teraz, że z tej ścieżki dostępu chcesz usunąć nazwę katalogu najniższego poziomu. Możesz to zrobić za pomocą znaków `%` i *, jak pokazuję w kolejnym przykładzie.

```
[donnie@fedora ~]$ echo ${pathname%/yum*}
/var/lib
[donnie@fedora ~]$
```

Znak `%` nakazuje powłoce pominięcie końcowego fragmentu ciągu tekstowego, który został dopasowany do wzorca. W tym przypadku gwiazdka na końcu nie jest konieczna, ponieważ ciąg tekstowy `yum` akurat znajduje się na końcu ścieżki. Zatem ten sam wynik uzyskasz także bez użycia gwiazdki. Jednak jeśli chcesz pominąć dwa najniższe poziomy ścieżki, musisz skorzystać z gwiazdki, aby dopasowanie wzorca odbyło się prawidłowo. Spójrz, oto co mi chodzi:

```
[donnie@fedora ~]$ echo ${pathname%/lib}
/var/lib/yum
[donnie@fedora ~]$ echo ${pathname%/lib*}
/var
[donnie@fedora ~]$
```

Uwaga

Jak widać, bez użycia gwiazdki dopasowanie wzorca nie zadziałało. Natomiast z gwiazdką wszystko działa poprawnie. Dlatego nawet jeśli gwiazdka nie jest absolutnie konieczna, najlepiej ją dodać dla pewności.

Z drugiej strony czasami może pojawić się potrzeba wyodrębnienia nazw katalogów niższego poziomu. Aby to zrobić, wystarczy zamienić znak `%` na `#` w następujący sposób:

```
[donnie@fedora ~]$ echo ${pathname#/var}
/lib/yum
[donnie@fedora ~]$ echo ${pathname#/var/lib}
/yum
[donnie@fedora ~]$
```

Na zakończenie tego punktu przedstawię jeszcze jedną ciekawą sztuczkę. Tym razem pokażę, jak dopasować wzorzec, a następnie podstawić coś innego. Najpierw utwórz zmienną `string` w następujący sposób:

```
[donnie@fedora ~]$ string="Hot and Spicy Food"
[donnie@fedora ~]$ echo $string
Hot and Spicy Food
[donnie@fedora ~]$
```

Wszystko działa dobrze, przy czym załóżmy, że nie chcesz umieszczać spacji między słowami. Zamiast tego użyjesz znaku podkreślenia (_), jak pokazuję w kolejnym przykładzie:

```
[donnie@fedora ~]$ echo ${string/[[:space:]]/_}
Hot_and Spicy Food
[donnie@fedora ~]$
```

To nie zadziałało najlepiej, ponieważ zastąpiona została tylko pierwsza spacja. Aby zastąpić wszystkie wystąpienia, po ciągu tekstowym musisz dodać kolejny ukośnik w taki sposób:

```
[donnie@fedora ~]$ echo ${string//[[:space:]]/_}
Hot_and_Spicy_Food
[donnie@fedora ~]$
```

To wygląda znacznie lepiej. Ale co tak naprawdę się tu dzieje? Cóż, używamy konstrukcji */wzorzec_do_zastąpienia/*, aby wykonać podstawienie. Cokolwiek umieścisz między dwoma ukośnikami, zostanie zastąpione. Możesz podać pojedynczy znak, klasę znaków lub inny wzorzec, który chcesz zastąpić. Na końcu, między ostatnim ukośnikiem i zamykającym nawiasem klamrowym, umieść znak, którym chcesz zastąpić wzorzec.

Wprawdzie na temat rozwijania zmiennych można powiedzieć znacznie więcej, ale tutaj zaprezentowałem tylko najbardziej praktyczne przykłady. Jeśli chcesz dowiedzieć się więcej, odnośniki do wartych uwagi zasobów znajdziesz na końcu rozdziału.

Skoro już omówiłem podstawianie wzorca, w następnym podrozdziale przejdę do podstawiania poleceń.

Podstawianie poleceń

We wcześniejszych ćwiczeniach praktycznych dotyczących zliczania zalogowanych użytkowników i korzystania z tablic pokazałem kilka przykładów zastosowania techniki podstawiania poleceń, ale jeszcze nie wyjaśniłem jej dokładnie. Najwyższy czas to nadrobić.

Podstawianie poleceń to niezwykle przydatne narzędzie, z którego będziesz często korzystać. Naprawdę. Za jego pomocą można zrobić wiele ciekawych rzeczy. Polega ono na wykorzystaniu wyniku polecenia powłoki w innym poleceniu lub przypisaniu go jako wartości zmiennej. Polecenie, którego wynik działania chcesz wykorzystać, umieszczasz w konstrukcji `$( )`. Oto prosty przykład:

```
[donnie@fedora ~]$ echo "This machine is running kernel version $(uname -r)."
This machine is running kernel version 6.5.5-200.fc38.x86_64.
[donnie@fedora ~]$
```

W tym przypadku wynik działania polecenia `uname -r`, które pokazuje wersję aktualnie uruchomionego jądra Linuksa, został użyty w miejsce konstrukcji podstawienia polecenia.

Teraz utwórz skrypt o nazwie *command_substitution_1.sh* i następującej zawartości:

```
#!/bin/bash
[[ ! -d Daily_Reports ]] && mkdir Daily_Reports
cd Daily_Reports
datestamp=$(date +%F)
echo "This is the report for $datestamp" > daily_report_$datestamp.txt
```

Wyjaśnię teraz sposób jego działania. W drugim wierszu skrypt sprawdza, czy istnieje katalog *Daily_Reports*. Jeśli go nie ma, zostanie utworzony. W czwartym wierszu konstrukcja podstawienia polecenia służy do utworzenia zmiennej `datestamp` razem z wartością w postaci bieżącej daty. Ta wartość jest zwracana przez polecenie `date +%F` i będzie zapisana w formacie rok-miesiąc-dzień (np. 2023-10-03). W ostatnim wierszu komunikat wraz z dzisiejszą datą zostaje zapisany do pliku, którego nazwa również zawiera tę datę. Spójrz na przykład użycia tego skryptu:

```
[donnie@fedora ~]$ ls -l Daily_Reports/
total 4
-rw-r--r--. 1 donnie donnie 34 Oct  3 15:30 daily_report_2023-10-03.txt
[donnie@fedora ~]$ cat Daily_Reports/daily_report_2023-10-03.txt
This is the report for 2023-10-03
[donnie@fedora ~]$
```

Niezłe, co? Uwierz mi, będziesz często stosować tego typu rozwiązania podczas tworzenia skryptów przeznaczonych do automatycznego generowania raportów.

Wskazówka

Polecenie `date` oferuje wiele różnych opcji formatowania. Aby się z nimi zapoznać, wystarczy wyświetlić stronę podręcznika systemowego dla tego polecenia (`man date`).

Ale chwileczkę, brakuje tutaj ważnego elementu. Co stanie się w sytuacji, gdy raport na dzisiaj został już utworzony? Czy chcesz go nadpisać?

Nie, nie w tym przypadku. Utwórz więc skrypt o nazwie *command_substitution_2.sh*, który zanim utworzy nowy raport, to najpierw sprawdzi, czy dzisiejszy raport już istnieje. Takie rozwiązanie wymaga dodania tylko niewielkiej liczby nowych poleceń, jak pokazuję w kolejnym fragmencie kodu:

```
#!/bin/bash
[[ ! -d Daily_Reports ]] && mkdir Daily_Reports
cd Daily_Reports
datestamp=$(date +%F)
[[ ! -f daily_report_$datestamp.txt ]] && echo "This is the report for
$datestamp" > daily_report_$datestamp.txt || echo "This report has already been
done today."
```

Uwaga

Ostatnie polecenie, które wygląda jak trzy wiersze, w rzeczywistości znajduje się w jednym wierszu zawijającym się na stronie drukowanej książki.

A teraz przygotuj się na to, co zobaczysz po uruchomieniu zmodyfikowanej wersji skryptu:

```
[donnie@fedora ~]$ ./command_substitution_2.sh
This report has already been done today.
[donnie@fedora ~]$
```

Dla zabawy przyjrzyj się kilku innym ciekawym przykładom.

Utwórz skrypt o nazwie *am_i_root_1.sh* i umieść w nim następujący kod:

```
#!/bin/bash
test $(whoami) != root && echo "You are not the root user."
test $(whoami) == root && echo "You are the root user."
```

Polecenie `whoami` zwraca nazwę użytkownika, który je uruchamia. Oto jak to wygląda, gdy uruchamiam je najpierw bez `sudo`, a następnie z `sudo`:

```
[donnie@fedora ~]$ whoami
donnie
[donnie@fedora ~]$ sudo whoami
root
[donnie@fedora ~]$
```

Jak widać, uruchomienie polecenia `whoami` z użyciem `sudo` pokazuje, że wykonującym je użytkownikiem jest `root`. Pierwsze polecenie wykorzystuje operatora `!=` do sprawdzenia, czy użytkownikiem nie jest `root`. Natomiast drugie polecenie używa operatora `==`, aby sprawdzić, czy użytkownikiem jest `root`. Teraz uruchom skrypt i zobacz, co się stanie:

```
[donnie@fedora ~]$ ./am_i_root_1.sh
You are not the root user.
[donnie@fedora ~]$ sudo ./am_i_root_1.sh
[sudo] password for donnie:
You are the root user.
[donnie@fedora ~]$
```

Skrypt działa, co oznacza, że jesteś już całkiem niezły w tworzeniu skryptów. Ale możesz być jeszcze lepszy, gdy uprościsz nieco kod. Zatem zmodyfikuj skrypt do następującej postaci:

```
#!/bin/bash
test $(whoami) != root && echo "You are not the root user." || echo "You are the
↪root user."
```

W tym skrypcie zamiast dwóch poleceń mamy tylko jedno. Jednak w obu przypadkach wynik jest identyczny.

Zamiast umieszczać polecenie w konstrukcji `$( )`, możesz również ująć je w apostrofy:

```
[donnie@fedora ~]$ datestamp=`date +%F`
[donnie@fedora ~]$ echo $datestamp
2023-10-03
[donnie@fedora ~]$
```

Wprawdzie ta metoda działa, ale jest przestarzała i nie polecam jej stosowania. Największy problem polega na tym, że jeśli dane polecenie zawiera znaki specjalne, które powłoka może błędnie zinterpretować, musisz pamiętać o ich odpowiednim zabezpieczeniu za pomocą ukośnika. Natomiast podczas używania nowszej konstrukcji `$( )` nie musisz się tym aż tak przejmować. Wspominałem o tej metodzie tylko dlatego, że możesz nadal natknąć się na skrypty, w których jest ona stosowana.

Na tym zakończę omawianie tematu podstawiania poleceń. W następnym podrozdziale dowiesz się, jak podejmować decyzje w kodzie skryptów.

Konstrukcje warunkowe i pętle

Dotychczas przedstawiałem wiele technik i konstrukcji programistycznych, które pozostają charakterystyczne dla skryptów powłoki. Natomiast w tym podrozdziale omówię kilka konstrukcji, które są powszechne w większości języków programowania. Rozpocznę od zaprezentowania innego sposobu podejmowania decyzji w programie.

Konstrukcja warunkowa if-then

Chociaż konstrukcje decyzyjne `&&` i `||` sprawdzają się w prostych skryptach, w przypadku bardziej złożonych operacji, takich jak testowanie wielu warunków jednocześnie, warto rozważyć użycie konstrukcji `if-then`. W pierwszym przykładzie utworzysz skrypt o nazwie *am_i_root_2.sh*, którego zawartość będzie przedstawiała się następująco:

```
#!/bin/bash
if [ $(id -u) == 0 ]; then
        echo "This user is root."
fi
if [ $(id -u) != 0 ]; then
        echo "This user is not root."
        echo "This user's name is $(id -un)."
fi
```

Uwaga

Zwróć uwagę, że każdy blok decyzyjny rozpoczyna się od słowa kluczowego `if` i kończy słowem kluczowym `fi` (tak, to `if` zapisane wspak). Warto również zaznaczyć, że — w przeciwieństwie do niektórych języków programowania — w skryptach powłoki bash wcięcia nie są wymagane. Niemniej jednak stosowanie wcięć znacznie poprawia czytelność kodu.

Zamiast używać polecenia `whoami`, tym razem korzystam z polecenia `id`, które oferuje więcej opcji. (Dodatkowe informacje na ich temat znajdziesz na stronach podręcznika systemowego dla obu poleceń). Jeśli chodzi o resztę, zamiast podejmować próbę szczegółowego wyjaśniania poszczególnych wierszy kodu, pozwolę Ci po prostu przestudiować ten skrypt i przekonać się samodzielnie, na czym polega jego działanie. Dla mnie to będzie łatwiejsze, zaś dla Ciebie mniej nużące. Poza tym — wierzę w Twoje umiejętności.

A teraz zobacz, co się stanie po uruchomieniu tego skryptu:

```
[donnie@fedora ~]$ ./am_i_root_2.sh
This user is not root.
This user's name is donnie.
[donnie@fedora ~]$
[donnie@fedora ~]$ sudo ./am_i_root_2.sh
This user is root.
[donnie@fedora ~]$
```

Gdy testujesz więcej niż tylko jeden warunek dla pewnej decyzji, bardziej właściwe jest użycie jednej konstrukcji `if-then-elif` zamiast dwóch oddzielnych `if-then`. Tym samym poprawisz czytelność kodu i dzięki temu każdy, kto go czyta, będzie mógł łatwiej zrozumieć sposób jego działania. Utwórz skrypt *am_i_root_3.sh*, w którym wykorzystasz tę technikę. Jego zawartość powinna przedstawiać się następująco:

```
#!/bin/bash
if [ $(id -u) == 0 ]; then
        echo "This user is root."
elif [ $(id -u) != 0 ]; then
        echo "This user is not root."
        echo "This user's name is $(id -un)."
fi
```

Słowo kluczowe `elif` jest tutaj skrótem od `else if`. Poza tym wszystko działa podobnie jak w poprzednim skrypcie. Po uruchomieniu otrzymasz dokładnie taki sam wynik jak wcześniej. Warto również zwrócić uwagę na możliwość sprawdzania wielu warunków za pomocą więcej niż tylko jednej klauzuli `elif`.

Ewentualnie możesz skorzystać z konstrukcji `if-then-else`. Utwórz skrypt *am_i_root_4.sh* o następującej zawartości:

```
#!/bin/bash
if [ $(id -u) == 0 ]; then
        echo "This user is root."
else
        echo "This user is not root."
        echo "This user's name is $(id -un)."
fi
```

Stosowanie klauzuli `else` może okazać się bardzo przydatne, ponieważ pozwala ona zdefiniować domyślne działanie, które zostanie podjęte, jeśli żaden z warunków w blokach `if` lub `elif` nie zostanie spełniony. Zapoznaj się teraz z przykładowym skryptem, który wykrywa system operacyjny uruchomiony w danym komputerze:

```
#!/bin/bash
os=$(uname)
if [[ $os == Linux ]]; then
        echo "This machine is running Linux."
elif [[ $os == Darwin ]]; then
        echo "This machine is running macOS."
elif [[ $os == FreeBSD ]]; then
        echo "This machine is running FreeBSD."
else
        echo "I don't know this $os operating system."
fi
```

Jak widać, ten skrypt potrafi wykrywać systemy operacyjne Linux, macOS lub FreeBSD. Jeśli komputer nie działa pod kontrolą żadnego z nich, wówczas klauzula `else` na końcu wyświetla domyślny komunikat. Warto również zauważyć, że na końcu poszczególnych poleceń `if` i `elif` należy umieścić średnik oraz słowo kluczowe `then`. To nie jest konieczne w przypadku polecenia `else`.

Spójrz na wynik uruchomienia omawianego skryptu w komputerze, który działa pod kontrolą systemu operacyjnego OpenIndiana:

```
donnie@openindiana:~$ ./os-test.sh
I don't know this SunOS operating system.
donnie@openindiana:~$
```

Oczywiście w tym skrypcie można dodać kolejną klauzulę `elif`, aby sprawdzić obecność systemu SunOS.

To w zasadzie wyczerpuje temat konstrukcji warunkowej `if-then`. W następnym punkcie przedstawię pętlę, która wykonuje określone operacje podczas oczekiwania na spełnienie pewnego warunku.

Konstrukcja do-while

Ta konstrukcja będzie wielokrotnie wykonywała zestaw poleceń, dopóki określony warunek pozostaje prawdziwy. Oto przykład jej użycia:

```
#!/bin/bash
x=10
while [[ $x -gt 0 ]]; do
        x=$(expr $x 1)
        echo $x
done
```

Ten skrypt *while_demo.sh* rozpoczyna działanie od przypisania wartości `10` zmiennej `x`. Dopóki wartość `x` pozostaje większa od `0`, skrypt odejmuje od niej `1` i przypisuje nową wartość zmiennej `x` za pomocą polecenia `expr $x-1`. Następnie wyświetla tę wartość. Wynik działania skryptu wygląda następująco:

```
[donnie@fedora ~]$ ./while_demo.sh
9
8
7
6
5
4
3
2
1
0
[donnie@fedora ~]$
```

Uwaga

Warto zauważyć, że w skrypcie *while_demo.sh* można zastosować skróconą notację do zmniejszania o `1` wartości zmiennej `x` w trakcie każdej iteracji pętli. W tym celu wystarczy wiersz `x=$(expr $x - 1)` zastąpić następującym:

```
((x--))
```

Jest to konstrukcja podobna do tej, którą można spotkać w programach napisanych w językach C lub C++. Należy jednak pamiętać, że nie jest ona przenośna, co oznacza, że działa poprawnie w powłoce bash, ale niekoniecznie już w innych. Dlatego jeśli chcesz, aby skrypt działał również w powłokach Bourne'a, dash lub ash, powinieneś unikać tej skróconej konstrukcji i pozostać przy poleceniu `x=$(expr $x - 1)`.

Pętli `while` możesz również użyć do odczytywania pliku tekstowego wiersz po wierszu. Oto prosty skrypt o nazwie *read_file.sh*, który odczytuje zawartość pliku */etc/passwd*:

```
#!/bin/bash
file=/etc/passwd
while read -r line; do
        echo $line
done < "$file"
```

Jak widzisz, skrypt rozpoczyna się od utworzenia zmiennej `file` i przypisania jej wartości */etc/passwd*. Wiersz `while` definiuje zmienną `line`, a polecenie `read -r` przypisuje wartości tej zmiennej. W trakcie każdej iteracji pętli `while` polecenie `read -r` odczytuje jeden wiersz pliku, przypisuje go zmiennej `line`, a następnie wyświetla ten wiersz na standardowym wyjściu (`stdout`). Po odczytaniu wszystkich wierszy pliku pętla kończy działanie. Ostatnie polecenie zawiera przekierowanie standardowego wejścia, aby pętla `while` odczytywała plik. Normalnie polecenie `read` dzieli długie wiersze na krótsze i kończy każdą część długiego wiersza ukośnikiem. Opcja `-r` wyłącza to zachowanie.

Możliwe są sytuacje, kiedy będziesz chciał utworzyć pętlę działającą w nieskończoność. Taką, która się nie zatrzyma, dopóki jej tego nie nakażesz. (Może też się zdarzyć, że przez przypadek zdefiniujesz tego rodzaju pętlę, ale to zupełnie inna historia. Na razie załóżmy, że chcesz to zrobić celowo). Aby przetestować tego rodzaju pętlę, utwórz skrypt *infinite_loop.sh* o następującej treści:

```
#!/bin/bash
while :
do
        echo "This loop is infinite."
        echo "It will keep going until you stop it."
        echo "To stop it, hit Ctrl-c."
        sleep 1
done
```

To dość bezużyteczny skrypt, którego działanie polega jedynie na wyświetlaniu kilku komunikatów. Polecenie `sleep 1` powoduje jednosekundowe opóźnienie między poszczególnymi iteracjami pętli. Oto co się stanie, gdy uruchomisz ten skrypt:

```
[donnie@fedora ~]$ ./infinite_loop.sh
This loop is infinite.
It will keep going until you stop it.
To stop it, hit Ctrl-c.
This loop is infinite.
It will keep going until you stop it.
To stop it, hit Ctrl-c.
^C
[donnie@fedora ~]$
```

Istnieje jeszcze kilka innych sztuczek, które możesz wykonać za pomocą pętli `while`, ale na razie to wystarczy. W następnym punkcie przyjrzysz się pętli `for-in`.

Konstrukcja for-in

Konstrukcja `for-in` przetwarza listę i wykonuje polecenie dla jej każdego elementu. W skrypcie *car_demo_2.sh* wiersz `for` tworzy zmienną `cars`. Spójrz na kod tego skryptu:

```
#!/bin/bash
for cars in Edsel Ford Nash Studebaker Packard Hudson
do
        echo "$cars"
done
echo "That's all, folks!"
```

W trakcie każdej iteracji pętli słowo kluczowe `in` pobiera nazwę klasycznego samochodu z listy i przypisuje ją jako wartość zmiennej `cars`. Pętla kończy działanie po przetworzeniu całej listy. Oto co się dzieje, gdy uruchomisz ten skrypt:

```
[donnie@fedora ~]$ ./car_demo_2.sh
Edsel
Ford
Nash
Studebaker
Packard
Hudson
That's all, folks!
[donnie@fedora ~]$
```

To dość proste, więc spróbuj czegoś innego. Tym razem utworzysz plik *list_demo.sh*:

```
#!/bin/bash
for filename in *
do
        echo "$filename"
done
```

Zadaniem tej pętli jest wyświetlenie plików, które znajdują się w bieżącym katalogu, czyli pętla działa podobnie do polecenia `ls`. Znak wieloznaczny * nakazuje pętli `for` odczytanie wszystkich nazw plików niezależnie od ich liczby. W poleceniu `echo` zmienna `$filename` została ujęta w cudzysłów na wypadek, gdyby któraś z nazw plików zawierała spacje. Oto co się dzieje, gdy uruchomisz ten skrypt:

```
[donnie@fedora ~]$ ./list_demo.sh
15827_zip.zip
2023-08-01_15-23-31.mp4
2023-08-01_16-26-12.mp4
2023-08-02_13-57-37.mp4
. . .
. . .
xargs_test.txt
yad-form.sh
zoneinfo.zip
[donnie@fedora ~]$
```

Dlaczego ten skrypt działa? Gdy polecenie `echo *` wydasz w wierszu poleceń, otrzymasz nieuporządkowaną listę plików w katalogu bieżącym. Pętla `for-in` powoduje, że nazwa każdego pliku jest wyświetlana w oddzielnym wierszu.

Na tym kończę omówienie pętli `for-in`. W następnym punkcie przyjrzysz się samej pętli `for`.

Konstrukcja for

Jest podobna do konstrukcji `for-in` z tą różnicą, że lista jest pobierana z innego źródła. W przypadku pętli `for` użytkownik podaje listę jako argument podczas wywoływania skryptu. Utwórz skrypt *car_demo_3.sh*, aby to przetestować:

```
#!/bin/bash
for cars
do
        echo "$cars"
done
```

Zmienna `cars` jest tworzona w wierszu, który zawiera słowo kluczowe `for`. Jednak nie ma w nim zdefiniowanej listy samochodów. Skąd więc bierze się ta lista? Jest tworzona na podstawie argumentów, które użytkownik wprowadza w powłoce podczas uruchamiania skryptu. Tym razem zamiast nazw klasycznych samochodów użyjesz listy współczesnych modeli, jak pokazuję w kolejnym przykładzie:

```
[donnie@fedora ~]$ ./car_demo_3.sh Toyota Volkswagen Subaru Honda
Toyota
Volkswagen
Subaru
Honda
[donnie@fedora ~]$
```

W kolejnym punkcie poznasz polecenie `break`.

Polecenie break

Użyj polecenia `break`, aby zachować większą kontrolę nad działaniem pętli `for-in` i `while-do`. Aby zobaczyć, jak to działa, utwórz skrypt *break_demo.sh* o następującej zawartości:

```
#!/bin/bash
j=0
while [[ $j -lt 5 ]]
do
        echo "This is number: $j"
        j=$((j + 1))
        if [[ "$j" == '2' ]]; then
                echo "We have reached our goal: $j"
                break
        fi
done
echo "That's all, folks!"
```

Polecenie `while` nakazuje skryptowi wykonywanie pętli tak długo, dopóki wartość zmiennej `j` jest mniejsza niż `5`. Konstrukcja `j=$((j + 1))` w szóstym wierszu to operator matematyczny, który zwiększa wartość zmiennej `j` o `1` w trakcie każdej iteracji pętli.

Konstrukcja `if-then`, która rozpoczyna się w siódmym wierszu, określa, co powinno się stać, gdy wartość zmiennej `j` jest równa `2`. W omawianym przykładzie polecenie `break` kończy wtedy działanie pętli. Spójrz na przykład uruchomienia tego skryptu:

```
[donnie@fedora ~]$ ./break_demo.sh
This is number: 0
This is number: 1
We have reached our goal: 2
That's all, folks!
[donnie@fedora ~]$
```

Uwaga

Jak już wspomniałem podczas omawiania skryptu *while_demo.sh*, istnieje możliwość zastąpienia wywołania `j=$((j + 1))` następującym skrótem:

```
((j++))
```

Jednak to jest skrót ściśle związany z powłoką bash i może nie działać w innych.

Wymienione polecenie można zapisać również w postaci `j=$(expr j + 1)` — ona także jest przenośna i stosuje postać, którą przedstawiłem podczas omawiania skryptu *while_demo.sh*.

(W rozdziale 11. przedstawię więcej informacji na temat przeprowadzania operacji matematycznych w skryptach powłoki).

Dla zabawy usuń polecenie `break` ze skryptu i uruchom go ponownie. Powinieneś otrzymać następujące dane wyjściowe:

```
[donnie@fedora ~]$ ./break_demo.sh
This is number: 0
This is number: 1
We have reached our goal: 2
This is number: 2
This is number: 3
This is number: 4
That's all, folks!
[donnie@fedora ~]$
```

Tym razem działanie pętli nie zatrzymuje się na liczbie `2`.

Skoro skończyliśmy `break`, możemy kontynuować.

Polecenie continue

Polecenie `continue` również modyfikuje działanie pętli `for-in` oraz `while-do`. Tym razem utwórz skrypt *for_continue.sh* o następującej zawartości:

```
#!/bin/bash
for cars in Pontiac Oldsmobile Buick Chevrolet Ford Mercury
do
        if [[ $cars == Buick || $cars == Mercury ]]; then
                continue
        fi
        echo $cars
done
```

W każdej iteracji pętli `for` inna nazwa klasycznego samochodu jest przypisywana zmiennej `cars`. Konstrukcja `if-then` sprawdza, czy wartość zmiennej `cars` to `Buick` lub `Mercury`. Polecenie `continue` w bloku `if-then` powoduje, że pętla pomija te dwie nazwy samochodów, więc polecenie `echo` ich nie wyświetli. W tym przykładzie widzisz również inne zastosowanie konstrukcji `||`. Gdy jest używana w warunku, wówczas działa jako operator logiczny *LUB*. Oto jak wygląda wynik działania tego kodu:

```
[donnie@fedora ~]$ ./for_continue.sh
Pontiac
Oldsmobile
Chevrolet
Ford
[donnie@fedora ~]$
```

Teraz spróbuj osiągnąć ten sam efekt przy użyciu pętli `while`. Utwórz skrypt o nazwie *while_continue.sh* i następującej zawartości:

```
#!/bin/bash
j=0
while [[ $j -lt 6 ]]
do
        j=$((j + 1))
        [[ $j -eq 3 || $j -eq 6 ]] && continue
        echo "$j"
done
```

Tym razem chcesz po prostu pominąć liczby `3` i `6`. Oto wynik uruchomienia skryptu:

```
[donnie@fedora ~]$ ./while_continue.sh
1
2
4
5
[donnie@fedora ~]$
```

Dobra, wystarczy tego. W następnym punkcie przejdę do pętli `until`.

Konstrukcja until

Pętla `until` będzie wykonywana do momentu spełnienia określonego warunku. Można ją wykorzystać na wiele sposobów, na przykład do opracowania gry w zgadywanie. Aby zobaczyć przykład jej zastosowania, utwórz skrypt o nazwie *secret_word.sh* i następującej zawartości:

```
#!/bin/bash
secretword=Donnie
word=
echo "Hi there, $USER!"
echo "Would you like to play a guessing game?"
echo "If so, then enter the correct secret word"
echo "to win a special prize."
echo
echo
until [[ "$word" = "$secretword" ]]
```

```
do
        echo -n "Enter your guess.  "
        read word
done
echo "Yay!  You win a pat on the back!"
```

Zmienna `secretWord` ma wartość `Donnie` (hej, to ja!). Z kolei zmienna `word` nie ma przypisanej żadnej wartości. Pętla `while` będzie wykonywana do momentu wprowadzenia poprawnej wartości zmiennej `secretWord`. (W tym przypadku polecenie `read` wstrzymuje działanie skryptu i czeka na wprowadzenie wartości przez użytkownika). Omówiony skrypt działa w następujący sposób:

```
[donnie@fedora ~]$ ./secret_word.sh
Hi there, donnie!
Would you like to play a guessing game?
If so, then enter the correct secret word
to win a special prize.
Enter your guess.  Vicky
Enter your guess.  Cleopatra
Enter your guess.  Donnie
Yay!  You win a pat on the back!
[donnie@fedora ~]$
```

Niezłe, prawda? To kolejna sztuczka, którą możesz zaimponować na najbliższej imprezie.

No dobrze, przechodzę do następnej konstrukcji.

Konstrukcja case

Konstrukcja `case` pozwala uniknąć stosowania konstrukcji `if-then-else`. Umożliwia użytkownikowi wprowadzenie ciągu tekstowego, a następnie sprawdza dostarczone dane i obsługuje opcję, którą wskazuje ten ciąg tekstowy. Spójrz na podstawową strukturę konstrukcji `case`:

```
case $variable in
  match_1)
     commands_to_execute
     ;;
  match_2)
   commands_to_execute
     ;;
  match_3)
   commands_to_execute
     ;;
*)  Optional Information
    commands_to_execute_for_no_match
     ;;
esac
```

Konstrukcja `case` jest porównywana z szeregiem wartości aż do znalezienia dopasowania. Wówczas wykonywane są wszystkie polecenia do podwójnego średnika (`;;`). Następnie rozpoczyna się wykonywane poleceń, które zostały zdefiniowane po wierszu polecenia `esac`.

Jeśli nie zostanie znalezione żadne dopasowanie, wykonywane są polecenia między *) i podwójnym średnikiem. Symbol *) działa podobnie jak klauzula `else` w konstrukcji `if-then` — definiuje domyślne działanie w przypadku, gdy żaden z testowanych warunków nie zostanie spełniony.

Dla zabawy wypróbuj tę konstrukcję. W tym celu utwórz skrypt *term_color.sh*, który będzie przedstawiał się następująco:

```
#!/bin/bash
echo -n "Choose Background Color for Terminal(b-black,g-grey): "
read color
case "$color" in
b)
  setterm -background black -foreground white
  ;;
g)
  setterm -background white -foreground black
  ;;
*)
  echo "I do not understand"
  ;;
esac
exit
```

Ten skrypt pozwala zmienić kolor tła terminala. (Doskonale wiem, że w przypadku opcji `g` zostało zdefiniowane tło w kolorze białym. To dlatego, że gdy uruchomisz ten skrypt i wybierzesz opcję `g`, tło będzie wyglądać bardziej na szare niż białe). Uruchomienie skryptu wygląda następująco:

```
[donnie@fedora ~]$ ./term_color.sh
Choose Background Color for Terminal(b-black,g-grey): g
[donnie@fedora ~]$
```

Uruchom skrypt w terminalu, wybierz opcję `g`, a powinieneś zobaczyć, jak tło powłoki zmienia kolor na szary. (Jeśli twój terminal ma już ustawione białe tło, wówczas wybierz opcję `b`). Aby zobaczyć, jak całe tło terminala zmienia się na szare, wystarczy wydać polecenie `clear`.

Dla jeszcze większej zabawy zmodyfikuj skrypt i dodaj kolejną opcję. Najpierw zmień wiersz polecenia `echo` na początku:

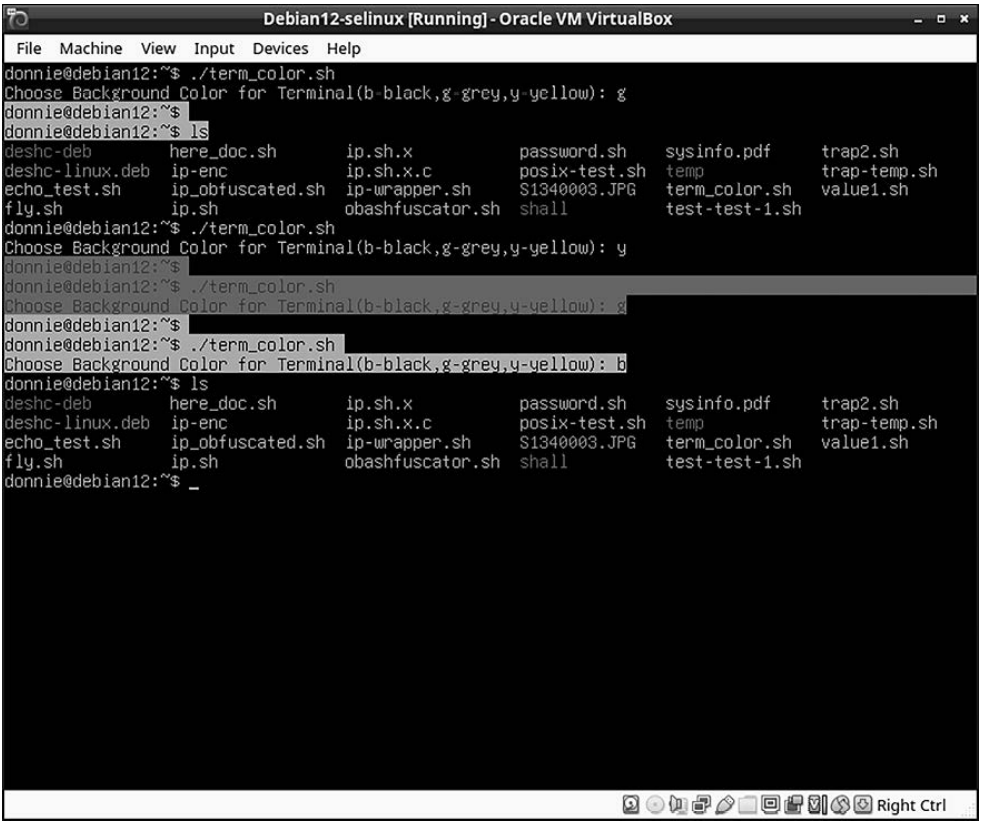
```
echo -n "Choose Background Color for Terminal(b-black,g-grey,y-yellow): "
```

Następnie po opcji `g` dodaj nową opcję `y`. Jej kod powinien przedstawiać się następująco:

```
y)
  setterm -background yellow -foreground red
  ;;
```

Aby zobaczyć coś naprawdę brzydkiego, uruchom skrypt ponownie i wybierz opcję `y`. (Nie martw się jednak, to ustawienie nie jest trwałe). Na rysunku 8.1 pokazuję, jak będzie wyglądać użycie różnych opcji.

Wcześniej już wyjaśniłem, jak używać pętli `for` do wprowadzania argumentów podczas uruchamiania skryptów. W następnym punkcie poznasz inny sposób.



Rysunek 8.1. Wynik działania skryptu term_color.sh

Używanie parametrów pozycyjnych

Podczas uruchamiania skryptu powłoki można również podać parametry, które będą wykorzystywane wewnątrz skryptu. Pierwszy podany parametr będzie oznaczony jako `$1`, drugi jako `$2` itd. (maksymalnie do `$9`). Parametr `$0` jest zarezerwowany dla nazwy samego skryptu.

Aby zobaczyć, jak to działa w praktyce, utwórz skrypt o nazwie *position_demo.sh*, którego zawartość będzie przedstawiała się następująco:

```
#!/bin/bash
# position_demo
echo "I have a cat, whose name is $1."
echo "I have another cat, whose name is $2."
echo "I have yet another cat, whose name is $3."
echo
echo
echo "The script that I just ran is $0"
```

Aby uruchomić skrypt, wpisz trzy parametry po nazwie skryptu w następujący sposób:

```
[donnie@fedora ~]$ ./position_demo.sh Vicky Cleopatra Lionel
```

```
I have a cat, whose name is Vicky.
I have another cat, whose name is Cleopatra.
I have yet another cat, whose name is Lionel.
The script that I just ran is ./position_demo.sh
[donnie@fedora ~]$
```

W danych wyjściowych skryptu zmienne `$1`, `$2` i `$3` zostaną zastąpione parametrami, które podałeś w powłoce. Zmienna `$0` zostanie rozwinięta do pełnej ścieżki i nazwy skryptu.

W skryptach możesz wykorzystać trzy specjalne parametry pozycyjne, które zwiększą funkcjonalność tworzonych skryptów. Oto ich lista:

- `$#`. Wyświetla liczbę wprowadzonych parametrów.
- `$@`. Wyświetla wszystkie wprowadzone parametry, każdy w osobnym wierszu.
- `$*`. Wyświetla wszystkie wprowadzone parametry w jednym wierszu, rozdzielone spacjami.

Ciekawą rzeczą, którą można zrobić za pomocą parametru `$#`, jest sprawdzanie błędów. Aby zobaczyć, co mam na myśli, uruchom ponownie skrypt *position_demo.sh*, ale tym razem podaj tylko jedno imię jako parametr. Powinieneś otrzymać dane wyjściowe podobne do następujących:

```
[donnie@fedora ~]$ ./position_demo.sh Vicky
I have a cat, whose name is Vicky.
I have another cat, whose name is .
I have yet another cat, whose name is .
The script that I just ran is ./position_demo.sh
[donnie@fedora ~]$
```

Jak widzisz, skrypt nie ostrzegł o tym, że nie podano poprawnej liczby parametrów. Zmodyfikuj go nieco, aby to naprawić. Nowa postać skryptu *position_demo_2.sh* przedstawia się więc następująco:

```
#!/bin/bash
# position_demo
if [[ $# -ne 3 ]]; then
        echo "This script requires three arguments."
        exit 1
fi
echo "I have a cat, whose name is $1."
echo "I have another cat, whose name is $2."
echo "I have yet another cat, whose name is $3."
echo
echo
echo "The script that I just ran is $0"
```

Uruchom go z trzema parametrami, a otrzymasz taki sam wynik jak w przypadku pierwszej wersji skryptu. Następnie uruchom go ponownie, ale tym razem podaj tylko jeden parametr. Teraz powinieneś zobaczyć coś takiego:

```
[donnie@fedora ~]$ vim position_demo_2.sh
[donnie@fedora ~]$ ./position_demo_2.sh Vicky
You entered 1 argument(s).
This script requires two arguments.
[donnie@fedora ~]$
```

To wygląda znacznie lepiej.

Zaprezentuję kolejną sztuczkę. Spójrz na wynik działania polecenia `date` bez określania żadnych opcji formatowania:

```
[donnie@fedora ~]$ date
Fri Oct  6 03:24:39 PM EDT 2023
[donnie@fedora ~]$
```

W wyniku wykonania polecenia `date` otrzymasz siedem następujących pól:

- dzień tygodnia,
- miesiąc,
- dzień miesiąca,
- godzina,
- przyrostek AM lub PM,
- strefa czasowa,
- rok.

Teraz utwórz skrypt *position_demo_3.sh*, który każde pole danych wyjściowych polecenia `date` będzie traktował jako parametr pozycyjny. Kod tego skryptu przedstawia się następująco:

```
#!/bin/bash
set $(date)
echo $*
echo "Day, First Argument: $1"
echo "Month, Second Argument: $2"
echo "Date, Third Argument: $3"
echo "Time, Fourth and Fifth Arguments: $4, $5"
echo "Time Zone, Sixth Argument: $6"
echo "Year, Seventh Argument: $7"
echo "$2 $3, $7"
```

W drugim wierszu pokazałem kolejne zastosowanie polecenia `set`, którego jeszcze nie omawiałem. Wcześniej było używane z opcją `-o` na potrzeby określania opcji powłoki. Tym razem użyjesz `set` bez żadnych opcji, ale z argumentem `$(date)`. Oto co mówi strona podręcznika systemowego powłoki bash na temat takiego użycia polecenia `set`:

> *Bez podania opcji polecenie wyświetla nazwy i wartości wszystkich zmiennych powłoki w formacie, który można wykorzystać jako dane wejściowe do ustawienia lub zresetowania aktualnie zdefiniowanych zmiennych.*

W tym przypadku polecenie `set` przetwarza wynik działania `$(date)` i formatuje go w sposób, który umożliwia wykorzystanie poszczególnych pól jako parametrów pozycyjnych.

W trzecim wierszu mamy do czynienia z prawdziwą magią. Parametr pozycyjny `$*` wyświetla w jednym wierszu wszystkie pola `$(date)`. Zadaniem pozostałych poleceń `echo` jest po prostu wyświetlenie tekstu, a następnie wartości określonego pola lub pól. Oto jak to wygląda w praktyce:

```
[donnie@fedora ~]$ ./position_demo_3.sh
Fri Oct 6 03:46:28 PM EDT 2023
Day, First Argument: Fri
```

```
Month, Second Argument: Oct
Date, Third Argument: 6
Time, Fourth and Fifth Arguments: 03:46:28, PM
Time Zone, Sixth Argument: EDT
Year, Seventh Argument: 2023
Oct 6, 2023
[donnie@fedora ~]$
```

Skrypt działa zgodnie z oczekiwaniami i wygląda całkiem fajnie. Dodaj go do listy sztuczek przeznaczonych do wypróbowania na następnej imprezie.

Myślę, że to wyczerpuje temat parametrów pozycyjnych. W następnym podrozdziale przejdę do kodów wyjścia.

Kody wyjścia

Prawdopodobnie widziałeś już kilka przykładów użycia polecenia `exit`, które może zakończyć działanie skryptu w normalny sposób lub spowodować jego wcześniejsze zakończenie w przypadku wystąpienia błędu. Czego jeszcze nie wyjaśniłem, to kwestii kodów wyjścia. Istnieją dwie ogólne kategorie kodów wyjścia:

- **Standardowe kody wyjścia powłoki**. Każda powłoka ma własny zestaw zdefiniowanych kodów wyjścia. (Dla uproszczenia w tym rozdziale omówię jedynie kody wyjścia stosowane w powłoce bash).
- **Niestandardowe kody wyjścia**. Istnieje możliwość samodzielnego zdefiniowania kodów wyjścia przeznaczonych do różnych celów.

Rozpocznę od omówienia standardowych kodów wyjścia.

Standardowe kody wyjścia powłoki

Gdy program lub skrypt kończą działanie z powodzeniem, zwracają kod wyjścia równy `0`. W przeciwnym razie kod wyjścia będzie liczbą niezerową z zakresu od `1` do `255`. Aby się o tym przekonać, użyj polecenia `find` do przeszukania katalogu */etc/* w celu znalezienia pliku *passwd*. Zrób to w następujący sposób:

```
[donnie@fedora ~]$ find /etc -name passwd
find: '/etc/audit': Permission denied
find: '/etc/cups/ssl': Permission denied
. . .
. . .
/etc/pam.d/passwd
find: '/etc/pki/rsyslog': Permission denied
find: '/etc/polkit-1/localauthority': Permission denied
find: '/etc/polkit-1/rules.d': Permission denied
. . .
. . .
find: '/etc/credstore.encrypted': Permission denied
/etc/passwd
[donnie@fedora ~]$
```

Jak widać, polecenie `find` znalazło plik, ale jednocześnie wygenerowało wiele błędów `Permission denied` (pol. odmowa dostępu) ze względu na to, że próbowało sprawdzić katalogi, do których nie ma dostępu użytkownik ze zwykłymi uprawnieniami. Teraz w następujący sposób sprawdź kod wyjścia tego polecenia:

```
[donnie@fedora ~]$ echo $?
1
[donnie@fedora ~]$
```

Zmienna specjalna `?` zwraca kod wyjścia ostatnio wykonanego polecenia. W tym przypadku wynosi on `1`, co oznacza, że wystąpił jakiś błąd. Konkretnie chodziło o to, że polecenie `find` nie uzyskało dostępu do niektórych katalogów, aby przeprowadzić w nich wyszukiwanie. Spróbuj więc ponownie, ale tym razem użyj polecenia `sudo`:

```
[donnie@fedora ~]$ sudo find /etc -name passwd
[sudo] password for donnie:
/etc/pam.d/passwd
/etc/passwd
[donnie@fedora ~]$ echo $?
0
[donnie@fedora ~]$
```

Tym razem otrzymujesz kod wyjścia `0`, co oznacza, że nie wystąpiły żadne błędy.

W większości przypadków zobaczysz kod wyjścia `0` lub `1`. Pełna lista kodów wyjścia, które możesz napotkać, obejmuje:

- `1` — błędy ogólne,
- `2` — nieprawidłowe użycie wbudowanych poleceń powłoki,
- `126` — brak możliwości wykonania żądanego polecenia,
- `127` — polecenie nie zostało znalezione,
- `128` — nieprawidłowy argument dla polecenia `exit`,
- `128+n` — krytyczny sygnał błędu `n`,
- `130` — skrypt przerwany przez naciśnięcie klawiszy *Ctrl+C*.

Można zademonstrować działanie niektórych innych kodów. Zacznij od utworzenia skryptu o nazwie *exit.sh* i następującej zawartości:

```
#!/bin/bash
exit n
```

Od razu widać błąd. Polecenie `exit` wymaga argumentu liczbowego i nie zadziała z argumentem tekstowym. Jednak udaj, że nie zauważyłeś tego błędu, i spróbuj uruchomić skrypt. Oto co otrzymasz w wyniku:

```
[donnie@fedora ~]$ ./exit.sh
./exit.sh: line 2: exit: n: numeric argument required
[donnie@fedora ~]$ echo $?
2
[donnie@fedora ~]$
```

Kod wyjścia `2` oznacza nieprawidłowe użycie **wbudowanego polecenia powłoki**.

Wspomniane **wbudowane polecenie powłoki** to po prostu polecenie, które nie ma własnego pliku wykonywalnego, ponieważ jest zintegrowane z plikiem wykonywalnym powłoki bash. Można pomyśleć, że w omawianej sytuacji powinien być wyświetlony kod wyjścia 128 z powodu podania nieprawidłowego argumentu dla polecenia `exit`. Jednak tak to nie działa. (Szczerze mówiąc, nie jestem pewien, co musiałbym zrobić, aby otrzymać kod 128. Ale to nie stanowi problemu). Aby zobaczyć pełną listę poleceń wbudowanych, wystarczy zajrzeć na stronę podręcznika systemowego, wyświetlaną po wydaniu polecenia `man builtins`.

Kod wyjścia 126 zwykle oznacza, że nie masz uprawnień do uruchomienia danego polecenia. Na przykład załóżmy, że zapomniałeś ustawić uprawnienia do wykonywania skryptu, co widać tutaj:

```
[donnie@fedora ~]$ ls -l somescript.sh
-rw-r--r--. 1 donnie donnie 0 Oct  7 16:26 somescript.sh
[donnie@fedora ~]$
```

Zobacz, co się stanie, gdy spróbujesz uruchomić ten skrypt:

```
[donnie@fedora ~]$ ./somescript.sh
bash: ./somescript.sh: Permission denied
[donnie@fedora ~]$ echo $?
126
[donnie@fedora ~]$
```

Kod wyjścia 127 zostanie wygenerowany, gdy spróbujesz wykonać nieistniejące polecenie:

```
[donnie@fedora ~]$ donnie
bash: donnie: command not found
[donnie@fedora ~]$ echo $?
127
[donnie@fedora ~]$
```

Moje imię oczywiście nie jest poprawnym poleceniem powłoki.

Kod 128+n oznacza, że wystąpił jakiś krytyczny błąd. Litera `n` oznacza dodatkową cyfrę dodawaną do 128. Na przykład jeśli uruchomisz polecenie i przerwiesz jego wykonywanie za pomocą kombinacji klawiszy *Ctrl+C*, otrzymasz kod 128+2, czyli 130. (Dwójka w tym przypadku wskazuje na konkretny rodzaj błędu o znaczeniu krytycznym).

W skryptach powłoki możesz używać standardowych kodów wyjścia do obsługi różnych sytuacji. Aby to zobaczyć w praktyce, utwórz skrypt o nazwie *netchk.sh* i następującej zawartości:

```
#!/bin/bash
if [[ $# -eq 0 ]]; then
        site="google.com"
else
        site="$1"
fi
ping -c 2 $site > /dev/null
if [[ $? != 0 ]]; then
        echo $(date +%F) . . . Network Failure!
        logger "Could not reach $site."
else
```

```
        echo $(date +%F) . . . Success!
        logger "$site is reachable."
fi
```

Ten skrypt oczekuje, że wywołasz go z argumentem w postaci nazwy hosta, nazwy domeny lub adresu IP. W pierwszej konstrukcji `if-then` na początku skryptu widać, że jeśli nie podasz argumentu, skrypt domyślnie użyje domeny *google.com*. W przeciwnym razie użyje podanego przez Ciebie argumentu. Następnie spróbuje wykonać polecenie `ping` do wskazanej domeny. Jeśli jego wykonanie się powiedzie, kod wyjścia będzie równy `0`. Jeśli się nie powiedzie — będzie to inna wartość.

W drugiej konstrukcji `if-then` widać, że jeśli kod wyjścia jest różny od `0`, skrypt wyświetli komunikat o błędzie sieci i umieści odpowiedni wpis w pliku dziennika systemowego (w systemie Fedora ten wpis trafi do pliku */var/log/messages*). W przeciwnym razie wyświetli komunikat o sukcesie. Oto jak wygląda przykładowy wynik uruchomienia skryptu:

```
[donnie@fedora ~]$ ./netchk.sh
2023-10-07 . . . Success!
[donnie@fedora ~]$ ./netchk.sh www.donnie.com
ping: www.donnie.com: Name or service not known
2023-10-07 . . . Network Failure!
[donnie@fedora ~]$
```

Niewiele więcej jest do powiedzenia na temat standardowych kodów wyjścia. Zatem w następnym punkcie poznasz kody wyjścia definiowane przez użytkownika.

Kody wyjścia zdefiniowane przez użytkownika

Można określić własne kody wyjścia — wystarczy podać argument liczbowy dla polecenia `exit`. Jest to przydatne, gdy zachodzi potrzeba przekazania konkretnego kodu wyjścia do programu zewnętrznego. Doskonałym przykładem jest narzędzie do monitorowania sieci Nagios.

Uwaga

Nagios to narzędzie, które potrafi monitorować praktycznie każdy rodzaj urządzenia w sieci. Może nadzorować różne typy serwerów, stacji roboczych, routerów, przełączników, a nawet drukarek. Tym, co czyni go tak wyjątkowym, jest modułowa konstrukcja, która umożliwia pracę z wtyczkami. Jeśli musisz monitorować konkretne urządzenie i okaże się, że nie ma odpowiedniej wtyczki, możesz po prostu przygotować własną. Wtyczki można tworzyć w różnych językach programowania. Mogą one mieć również postać skryptów powłoki.

Na serwerze lub stacji roboczej, którą chcesz monitorować, możesz zainstalować agenta monitorującego Nagios i utworzyć skrypt powłoki do generowania kodów wyjścia oczekiwanych przez Nagios. Aby poznać sposób działania takiego rozwiązania, przyjrzyj się poniższemu fragmentowi kodu, który jest częścią większego skryptu:

```
#!/bin/bash
os=$(uname)
```

```
quantity=$(cut -f3 -d: /etc/passwd | grep -w 0 | wc -l)
if [ $os == Linux ]; then
        if [ $quantity -gt 1 ]; then
                echo "CRITICAL.  There are $quantity accounts with UID 0."
                exit 2
        else
                echo "OKAY.  There is only one account with UID 0."
                exit 1
        fi
```

Ten skrypt analizuje plik */etc/passwd*, aby sprawdzić, czy istnieje więcej niż jeden użytkownik z identyfikatorem UID równym `0`. Jest to istotne, ponieważ wymieniony identyfikator nadaje kontu uprawnienia użytkownika `root`. W systemie Linux nigdy nie powinno być więcej niż jedno konto z identyfikatorem UID o wartości `0`. W kodzie konstrukcji `if-then` widać, że jeśli skrypt znajdzie więcej niż jedno takie konto, wówczas wygeneruje kod wyjścia `2`. W przeciwnym razie wygeneruje kod wyjścia `1`.

Ten kod wyjścia wraz z odpowiadającym mu poleceniem `echo` są przekazywane do agenta monitorującego Nagios. Następnie ten agent przekaże dane wyjściowe polecenia `echo` do serwera Nagios, który z kolei wyświetli komunikat w panelu Nagios. (Cały skrypt przedstawię w dalszej części rozdziału).

To w zasadzie wszystko, jeśli chodzi o kody wyjścia. W następnym podrozdziale dowiesz się nieco więcej na temat polecenia `echo`.

Więcej informacji o poleceniu echo

Już poznałeś najprostszy sposób użycia polecenia `echo`, które służy do wyświetlania komunikatów na ekranie lub zapisywania tekstu do pliku. Teraz przyjrzysz się różnym opcjom formatowania dostępnym dla tego polecenia.

Jeśli użyjesz opcji `-n`, zapobiegniesz tworzeniu nowego wiersza na końcu wyświetlanego tekstu. Spójrz na poniższy przykład:

```
[donnie@fedora ~]$ echo -n "The fat cat jumped over the skinny dog."
The fat cat jumped over the skinny dog.[donnie@fedora ~]$
```

Tej opcji użyj w połączeniu z `-e`, a zyskasz dostęp do opcji, które wykorzystują ukośnik. Na przykład jeśli chcesz wstawić pionowy znak tabulacji do wiersza tekstu, użyj opcji `-e` w połączeniu z `\v` w następujący sposób:

```
[donnie@fedora ~]$ echo -e "The fat cat jumped\v over the skinny dog."
The fat cat jumped
                    over the skinny dog.
[donnie@fedora ~]$
```

Aby wstawić tabulator poziomy, użyj opcji `\t`, na przykład tak:

```
[donnie@fedora ~]$ echo -e "The fat cat jumped\t over the skinny dog."
The fat cat jumped        over the skinny dog.
[donnie@fedora ~]$
```

Natomiast jeśli chcesz wstawić ukośnik \ do tekstu, użyj po prostu dwóch kolejnych ukośników w ten sposób:

```
[donnie@fedora ~]$ echo -e "The fat cat jumped over the thin\\skinny dog."
The fat cat jumped over the thin\skinny dog.
[donnie@fedora ~]$
```

Nie jesteś ograniczony tylko do wyświetlania wiadomości tekstowych. Możesz również użyć znaku wieloznacznego, aby wyświetlić listę plików znajdujących się w bieżącym katalogu, jak pokazuję w kolejnym przykładzie:

```
[donnie@fedora ~]$ echo *
1 15827_zip.zip 18.csv 2023-08-01_15-23-31.mp4 2023-08-01_16-26-12.mp4 2023-
08-02_13-57-37.mp4 2023-10-25_price.txt 21261.zip 4-2_Building_an_Alpine_
Container.bak 4-2_Building_an_Alpine_Container.pptx 46523.zip 48986.zip 50645.
zip 54586.zip 70604.zip access_log_parse.sh access_log_parse.txt actorfile_10.
txt actorfile_11.txt actorfile_1.txt actorfile_2.txt actorfile_4.txt
actorfile_5.txt actorfile_6.txt actorfile_7.txt actorfile_8.txt actorfile_9.
txt add_fields.awk add-repos.sh addresses.txt alignment_1.txt alignment_2.txt
alma9_default.txt alma9_future.txt alma_link.t
. . .
. . .
donnie@fedora:~$
```

Możesz również wyświetlić komunikat wraz z listą plików, na przykład w taki sposób:

```
[donnie@fedora ~]$ echo -e "These are my files:\n" *
These are my files:
 15827_zip.zip 2023-08-01_15-23-31.mp4 2023-08-01_16-26-12.mp4
. . .
. . .
test.txt yad-form.sh zoneinfo.zip
[donnie@fedora ~]$
```

Przy odrobinie wyobraźni będziesz w stanie wykorzystać te opcje formatowania, aby poprawić wygląd danych wyświetlanych na ekranie oraz dokumentów tekstowych.

Uwaga

Niestety, mimo że te opcje formatowania dla polecenia `echo` są bardzo ciekawe, nie działają one dobrze w niektórych powłokach innych niż bash, na przykład w dash. W rozdziale 19., poświęconym przenośności skryptów powłoki, pokażę, jak rozwiązać ten problem za pomocą polecenia `printf` zamiast `echo`.

To by było na tyle, jeśli chodzi o polecenie `echo`. W następnym podrozdziale przejdziesz do rzeczywistych przykładów, w których użyto omówione wcześniej techniki.

Kilka rzeczywistych przykładów omówionych technik

W tej części pokażę kilka praktycznych zastosowań dla technik, które dotychczas zostały omówione. A właściwie to zamiast tylko pokazywać, pozwolę Ci samodzielnie wypróbować je w ciekawych i praktycznych ćwiczeniach.

Ćwiczenie praktyczne — stosowanie konstrukcji if-then

To jest prawdziwy przykład z życia wzięty. Kilka lat temu opracowałem ten skrypt jako wtyczkę do systemu monitorowania sieci Nagios. Chodziło o to, aby upewnić się, że złośliwi hakerzy nie dodali nieuprawnionego konta z identyfikatorem UID `0` do pliku */etc/passwd* w komputerach z systemami Linux i FreeBSD. Dzieje się tak, ponieważ każde konto z identyfikatorem UID o wartości `0` w pliku *passwd* ma pełne uprawnienia użytkownika `root`. Na pewno nie chcesz, aby jakiekolwiek nieautoryzowane konta miały takie uprawnienia.

Problem polega na tym, że w komputerach z Linuksem powinno być tylko jedno konto użytkownika o identyfikatorze UID `0`, zaś w systemie FreeBSD są dwa takie konta. (Jedno nosi nazwę `toor` i ma ustawioną powłokę bash jako domyślną. Drugie konto to `root`, które ma ustawioną powłokę csh jako domyślną). Potrzebny jest więc skrypt, który będzie działał w obu tych systemach operacyjnych. (Pamiętaj, że podczas tego ćwiczenia będziesz modyfikować plik *passwd*, więc najlepiej jest wykonać je w maszynie wirtualnej, a nie na rzeczywistym serwerze produkcyjnym).

Zwróć uwagę, że kody wyjścia `1` i `2`, które otrzymasz podczas wykonywania skryptu, są oczekiwane przez Nagios, aby wskazać stan `OKAY` lub `CRITICAL`. Pamiętaj też, że możesz dodać więcej bloków `elif`, jeśli chcesz sprawdzać inne systemy UNIX lub podobne. (W rzeczywistości zobaczysz, że właśnie dodałem kod przeznaczony do sprawdzania systemów macOS i OpenIndiana). Po tym wstępie przechodzimy do skryptu.

1. Niestety skrypt jest zbyt długi, aby go tutaj w całości przedstawić. Dlatego przejdź do repozytorium z materiałami do książki w serwisie GitHub i pobierz skrypt *UID-0_check.sh*. Przenieś go do maszyny wirtualnej z Linuksem. Otwórz skrypt w edytorze tekstu i przeanalizuj kod.
2. Uruchom skrypt, aby zobaczyć wygenerowane dane wyjściowe. Powinieneś zobaczyć następujący komunikat:

   ```
   [donnie@fedora ~]$ ./UID-0_check.sh
   OKAY.  There is only one account with UID 0.
   [donnie@fedora ~]$
   ```

3. **OSTRZEŻENIE: Jak wcześniej wspomniałem, to ćwiczenie należy wykonać w maszynie wirtualnej, a nie w komputerze produkcyjnym.**

   W maszynie wirtualnej z systemem Linux utwórz nowe konto użytkownika. W tym celu skorzystaj z odpowiedniego polecenia tworzenia użytkowników

w danej dystrybucji Linuksa. Otwórz plik */etc/passwd* w edytorze tekstu i zmień wartość identyfikatora UID nowego użytkownika na `0`.

Pole identyfikatora UID jest trzecim polem w każdym wierszu pliku *passwd*. Na przykład w tym wierszu identyfikator UID użytkownika `Vicky` wynosi `1001`:

```
vicky:x:1001:1001::/home/vicky:/bin/bash
```

Zmiana tej wartości na `0` spowoduje, że wiersz będzie przedstawiał się następująco:

```
vicky:x:0:1001::/home/vicky:/bin/bash
```

4. Zapisz plik i uruchom skrypt ponownie. Teraz powinieneś zobaczyć komunikat podobny do następującego:

```
[donnie@fedora ~]$ ./UID-0_check.sh
CRITICAL.  There are 2 accounts with UID 0.
[donnie@fedora ~]$
```

5. Usuń nowo utworzone konto użytkownika.
6. Utwórz maszynę wirtualną z systemem FreeBSD i zainstaluj w niej pakiety `sudo` oraz `bash`, tak jak to wyjaśniłem na początku książki. Skopiuj skrypt *UID-0_check.sh* do tej maszyny i powtórz kroki od 3. do 5. Tym razem dwa konta są oznaczone jako `OKAY`, natomiast trzy jako `CRITICAL`. W ten sposób masz potwierdzenie, że fragment kodu `elif [ $os == FreeBSD ]; then` na końcu skryptu poprawnie wykrywa system operacyjny i wykonuje przeznaczony dla niego kod.

Koniec ćwiczenia.

Ćwiczenie praktyczne — analiza dziennika dostępu serwera Apache

W tym ćwiczeniu pokażę, jak potężny może być jednowierszowy skrypt powłoki. Jednak opracowanie takiego polecenia może być dość skomplikowane i dlatego wyjaśnię, jak je zbudować krok po kroku. Na każdym etapie pracy upewnisz się o poprawności jego działania i dopiero wtedy przejdziesz do następnego etapu. Jeśli jesteś gotowy, zaczynajmy.

1. Przygotuj maszynę wirtualną z systemem Fedora Server, w której jest użyta sieć mostkowa. (Ten rodzaj sieci będzie potrzebny, aby uzyskać dostęp do maszyny wirtualnej z poziomu innych urządzeń, które znajdują się w Twojej sieci).
2. Zainstaluj i uruchom serwer WWW Apache w następujący sposób:

```
$ sudo dnf install httpd
$ sudo systemctl enable --now httpd
```

3. Otwórz port 80 w zaporze sieciowej maszyny wirtualnej w następujący sposób:

```
$ sudo firewall-cmd --permanent --add-service=http
$ sudo firewall-cmd --reload
```

4. W jak największej liczbie innych urządzeń w Twojej sieci uruchom przeglądarkę internetową i wpisz adres IP maszyny wirtualnej. Ten adres powinien wyglądać mniej więcej tak:

   ```
   http://192.168.0.10
   ```

   Pamiętaj, że możesz uzyskać dostęp z poziomu zarówno fizycznych komputerów, jak i innych maszyn wirtualnych, które znajdują się w tej samej sieci. Warto również zaznaczyć, że nie ma potrzeby tworzenia własnej strony internetowej, ponieważ domyślna *strona testowa serwera Fedora* będzie wystarczająca na potrzeby tego ćwiczenia.

5. Wyświetl dziennik dostępu serwera Apache za pomocą następującego polecenia:

   ```
   $ sudo less /var/log/httpd/access_log
   ```

   Zwróć uwagę, że każdy wiersz rozpoczyna się od adresu IP maszyny, która uzyskała dostęp do tej strony internetowej. Oto przykład:

   ```
   192.168.0.25 [06/Oct/2023:16:44:15 -0400] "GET /poweredby.png
   HTTP/1.1" 200 5714 "http://192.168.0.10/" "Mozilla/5.0 (Windows NT 10.0;
   Win64; x64; rv:109.0) Gecko/20100101 Firefox/118.0"
   ```

6. Jak widać, źródłowy adres IP znajduje się w pierwszym polu, a poszczególne pola są oddzielone spacjami. Aby wyświetlić tylko listę źródłowych adresów IP, możesz użyć polecenia `cut` — w jego wywołaniu wskaż spację jako separator i wybierz tylko pierwsze pole. Polecenie i wynik jego wywołania wyglądałyby mniej więcej tak:

   ```
   [donnie@fedora-server ~]$ sudo cut -d" " -f1 /var/log/httpd/access_log
   ::1
   192.168.0.16
   192.168.0.16
   192.168.0.27
   . . .
   . . .
   192.168.0.25
   192.168.0.25
   192.168.0.9
   192.168.0.8
   192.168.0.8
   192.168.0.8
   192.168.0.8
   [donnie@fedora-server ~]$
   ```

   Otrzymasz listę adresów IPv4 maszyn, które uzyskały dostęp do tego serwera, z jednym wyjątkiem. Tym wyjątkiem jest adres IPv6 na początku listy, który jest adresem `localhost` komputera działającego pod kontrolą systemu Fedora Server. (Prawdopodobnie nie zobaczysz tego adresu IPv6, chyba że wyświetlisz stronę testową z poziomu samej maszyny wirtualnej).

7. Do tej pory wszystko idzie dobrze. Udało się wyodrębnić pierwsze pole z danych dziennika serwera Apache. Teraz dodasz drugą część polecenia, która posortuje dane wyjściowe, aby filtr `uniq` zadziałał poprawnie w następnym kroku. Oto jak to wygląda zmodyfikowana wersja polecenia:

   ```
   [donnie@fedora-server ~]$ sudo cut -d" " -f1 /var/log/httpd/access_log | sort
   ```

Jeżeli użyjesz polecenia `sort` bez opcji `-n`, lista nie zostanie posortowana w odpowiedniej kolejności liczbowej. Jednak na tym etapie nie ma to znaczenia.

8. Kolejnym krokiem jest usunięcie z danych wyjściowych powtarzających się adresów IP oraz zliczenie, ile razy każdy z nich występuje w pliku źródłowym. Oto jak wygląda zmodyfikowane polecenie:

```
[donnie@fedora-server ~]$ sudo cut -d" " -f1 /var/log/httpd/access_log | sort |
↪uniq -c
      1 ::1
     11 192.168.0.16
      4 192.168.0.25
      4 192.168.0.27
      4 192.168.0.8
      1 192.168.0.9
[donnie@fedora-server ~]$
```

9. Teraz przeprowadzisz sortowanie według liczby wystąpień każdego adresu IP w odwrotnej kolejności liczbowej. Zrobisz to w następujący sposób:

```
[donnie@fedora-server ~]$ sudo cut -d" " -f1 /var/log/httpd/access_log | sort |
↪uniq -c | sort -nr
     11 192.168.0.16
      4 192.168.0.8
      4 192.168.0.27
      4 192.168.0.25
      1 192.168.0.9
      1 ::1
[donnie@fedora-server ~]$
```

10. Skoro masz pewność, że polecenie działa poprawnie, utwórz skrypt o nazwie *ipaddress_count.sh* i następującej zawartości:

```
#!/bin/bash
cut -d" " -f1 /var/log/httpd/access_log | sort | uniq -c | sort -nr
```

Pamiętaj, że do jego uruchomienia są wymagane uprawnienia administratora (`sudo`).

11. Na koniec nieco ulepszysz przykładowy skrypt. Dodaj kod, który zapisze dane wyjściowe do pliku tekstowego z nazwą zawierającą znacznik czasu. Gotowy skrypt będzie przedstawiał się następująco:

```
#!/bin/bash
timestamp=$(date +%F)
echo "These addresses have accessed this webserver as of $timestamp." >
ipaddress_list_$timestamp.txt
cut -d" " -f1 /var/log/httpd/access_log | sort | uniq -c | sort -nr >>
ipaddress_list_$timestamp.txt
```

Oczywiście istnieją inne programy, które wykonują bardziej kompleksową analizę plików dziennika serwera WWW. Jednak dzięki omówionemu tutaj skryptowi możesz szybko sprawdzić, kto uzyskuje dostęp do Twojego serwera.

Koniec ćwiczenia.

Możesz teraz przejść do ostatniego ćwiczenia.

Ćwiczenie praktyczne — testy beta nowego dysku twardego

Ostatni przykład dotyczy doświadczenia, które miałem kilka lat temu. Wtedy to firma Western Digital zaprosiła mnie do udziału w testach beta nowego modelu dysku twardego. Moim zadaniem było jedynie utrzymanie dysku w ciągłym działaniu przez cały czteromiesięczny okres testów, a następnie zebranie danych z dziennika BIOS-u dysku po zakończeniu testów. Jednak postanowiłem pójść o krok dalej i napisałem skrypt powłoki, który codziennie automatycznie zbierał dane o wydajności dysku. Podobnie jak w przypadku poprzednich ćwiczeń także to wykonaj w swojej maszynie wirtualnej z systemem Fedora Server.

1. Aby zbierać dane o wydajności dysku, musisz zainstalować kilka pakietów. Zrobisz to za pomocą następującego polecenia:

   ```
   $ sudo dnf install sysstat smartmontools
   ```

2. Uruchom usługę `sysstat` i upewnij się, że jest aktywna. W tym celu wydaj przedstawione tutaj polecenia:

   ```
   $ sudo systemctl start sysstat
   $ systemctl status sysstat
   ```

3. Do zbierania danych wykorzystasz komponent `sar` z pakietu `sysstat`. Jednak zanim jakiekolwiek dane będą dostępne, minie kilka minut. W międzyczasie możesz wygenerować trochę aktywności na dysku twardym poprzez przeprowadzenie aktualizacji systemu. Zrób to w następujący sposób:

   ```
   $ sudo dnf -y upgrade
   ```

4. Zapoznaj się ze stroną podręcznika systemowego `man` dla polecenia `sar` i zwróć uwagę na typy danych, które można zbierać za pomocą różnych jego opcji. Niektóre z tych opcji zostały użyte w skrypcie powłoki.

   Ten skrypt jest zbyt obszerny, aby go tutaj w całości przedstawić. Dlatego pobierz plik *hard_drive.sh* z repozytorium w serwisie GitHub. Otwórz plik w edytorze tekstu i przeanalizuj. Wszystkie koncepcje użyte w tym skrypcie zostały już omówione, więc powinieneś być w stanie zrozumieć sposób jego działania.

5. Ostatnie polecenie w skrypcie to `smartctl` i wymaga ono uprawnień `sudo`. Dlatego będziesz musiał użyć `sudo`, aby uruchomić skrypt, na przykład w taki sposób:

   ```
   $ sudo ./hard_drive.sh
   ```

   Uzbrój się w cierpliwość, ponieważ wykonanie tego zadania zajmie kilka minut. Pamiętaj też, że wirtualny dysk maszyny wirtualnej nie jest rozpoznawany przez `smartmontools`. Dlatego w raporcie pojawią się pewne komunikaty ostrzegawcze.

6. Po zakończeniu działania skryptu zajrzyj do raportu, który został wygenerowany w katalogu *Drive_Reports*.

7. Możesz również uruchomić ten skrypt w swoim fizycznym komputerze z Linuksem. Powinien działać na większości dystrybucji, o ile masz zainstalowane pakiety `sysstat` i `smartmontools` oraz uruchomioną usługę `sysstat`.

Koniec ćwiczenia.

To już wszystko w tym rozdziale. Poniżej znajdziesz krótkie podsumowanie, a następnie możesz przejść do następnego rozdziału.

Podsumowanie

W tym rozdziale omówiłem ogromną ilość materiału i mam nadzieję, że Cię nim nie przytłoczyłem. Moim celem było przedstawienie kompleksowego przeglądu koncepcji i technik, których można użyć do tworzenia funkcjonalnych skryptów powłoki. Zacząłem od wyjaśnienia technik charakterystycznych dla skryptów powłoki, a następnie przeszedłem do technik wspólnych dla większości języków programowania.

Tak naprawdę to jeden z najfajniejszych aspektów poznawania tematu tworzenia skryptów powłoki. Jest to o wiele łatwiejsze do opanowania niż języki programowania wyższego poziomu — takie jak C, Java czy Rust — a jednocześnie niezwykle przydatne. Co więcej, ucząc się skryptów powłoki, poznajesz również konstrukcje i koncepcje, które mają zastosowanie w tych językach. Dlatego jeśli planujesz w przyszłości nauczyć się innego języka programowania, nauka skryptów powłoki może być świetnym przygotowaniem.

Pomimo że przedstawiłem ogromną ilość materiału, to jeszcze nie koniec. W następnym rozdziale zaprezentuję kilka dodatkowych sposobów filtrowania i przeprowadzania operacji na tekście. Do zobaczenia!

Pytania

1. Który z poniższych fragmentów kodu przedstawia najczęściej preferowany sposób podstawiania poleceń?
   - **A.** `` `polecenie` ``
   - **B.** `%(polecenie)`
   - **C.** `"polecenie"`
   - **D.** `$(polecenie)`
2. Jak utworzyć tablicę imion?
   - **A.** `set array=names`

```
names=( Vicky Frank Cleopatra Katelyn )
```

   - **B.** `array=names`

```
names=( Vicky Frank Cleopatra Katelyn )
```

   - **C.** `array names`

```
names=( Vicky Frank Cleopatra Katelyn )
```

   - **D.** `declare names`

```
names=( Vicky Frank Cleopatra Katelyn )
```

E. `declare -a names`

```
names=( Vicky Frank Kleopatra Katelyn )
```

3. Jak sprawdzić kod wyjścia ostatnio wykonanego polecenia?
   A. `echo $#`
   B. `echo $?`
   C. `echo $$`
   D. `echo $!`
4. Chcesz zdefiniować pętlę, która odczyta listę imion, a następnie zapisze te imiona do innego pliku tekstowego. Jednak chcesz pominąć dwa z tych imion. Które z poniższych poleceń spowoduje, że Twój skrypt zrobi to poprawnie?
   A. `break`
   B. `skip`
   C. `continue`
   D. `stop`
5. Chcesz porównać dwie wartości liczbowe, aby sprawdzić, czy są równe. Którego z następujących operatorów użyjesz?
   A. `=`
   B. `-eq`
   C. `==`
   D. `-ne`

Lektura uzupełniająca

- Artykuł *What is the Bash Shebang and How to Use it* na stronie *https://www.rosehosting.com/blog/what-is-the-bash-shebang/*.
- Artykuł *An introduction to parameter expansion in Bash* na stronie *https://opensource.com/article/17/6/bash-parameter-expansion*.
- Sekcja *Shell Parameter Expansion* (w podręczniku powłoki bash) na stronie *https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html*.
- Artykuł *Introduction to if* na stronie *https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_07_01.html*.
- Artykuł *Bash while Loop* na stronie *https://linuxize.com/post/bash-while-loop/*.
- Artykuł *How to Find Most Used Disk Space Directories and Files in Linux* na stronie *https://www.tecmint.com/find-top-large-directories-and-files-sizes-in-linux/*.
- Artykuł *Standard Exit Status Codes in Linux* na stronie *https://www.baeldung.com/linux/status-codes*.
- Artykuł *How to Use the sar Command on Linux* na stronie *https://www.howtogeek.com/793513/how-to-use-the-sar-command-on-linux/*.

Odpowiedzi

1. d
2. e
3. b
4. c
5. b

Skorowidz

G

H

I

J

K

L

M

N

O

P

R

S

W

Z

Poświęć skryptowi minutę, a zaoszczędzisz godziny!

Wiele zadań administracyjnych w Linuksie można wykonać, korzystając z graficznego interfejsu użytkownika. Jednak prawdziwą moc swojego systemu uwolnisz za pomocą skryptów wiersza poleceń i powłoki. W ten sposób możesz efektywnie automatyzować powtarzalne zadania, precyzyjnie skonfigurować system, a także zapewnić wysoki stopień bezpieczeństwa.

Ta praktyczna książka świetnie posłuży początkującym i bardziej zaawansowanym administratorom Linuksa. Będzie też pomocna w przygotowaniu się do egzaminów CompTIA Linux+ i Linux Professional Institute. Rozpoczniesz od podstaw korzystania z powłoki, aby w kolejnych rozdziałach przejść do bardziej zaawansowanych koncepcji. Zobaczysz, jak tworzyć skrypty automatyzujące powtarzalne zadania administracyjne, a także wiele innych przydatnych rozwiązań. W książce znajdziesz praktyczne, gotowe do użycia skrypty. Zostały one opracowane w taki sposób, by ułatwić zarządzanie systemem, wspomagać naukę omawianych koncepcji i pomagać podczas rozwiązywania problemów. Przede wszystkim skoncentrujesz się na powłoce bash, ale zapoznasz się również z powłoką Zsh i PowerShell.

W książce:

- koncepcja powłoki i różne rodzaje powłok
- przekierowanie, potok i polecenia złożone
- filtry strumieni tekstowych i dynamiczne przetwarzanie danych
- funkcje, biblioteki i tworzenie modułowych skryptów powłoki
- struktura przenośnych skryptów powłoki

Donald A. Tevault od 2006 roku pracuje z systemami Linux. Posiada certyfikaty Level 3 Security i GIAC Incident Handler, zajmował się bezpieczeństwem Linuksa w kontekście internetu rzeczy. Jest profesjonalnym wykładowcą wiedzy o Linuksie. Obecnie prowadzi kanał BeginLinux Guru w serwisie YouTube. Jest również autorem książki *Bezpieczeństwo systemu Linux. Hardening i najnowsze techniki zabezpieczania przed cyberatakami.*

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-3175-6
9 788328 931756
Cena: 149,00 zł
packt