Mokhtar Ebrahim
Andrew Mallett

Skrypty powłoki systemu Linux

Zagadnienia zaawansowane

Wydanie II

Helion
Packt

Tytuł oryginału: Mastering Linux Shell Scripting - Second Edition

Tłumaczenie: Grzegorz Kowalczyk

ISBN: 978-83-283-5070-0

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/skrzz2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

ROZDZIAŁ 1

Co i dlaczego w skryptach powłoki bash

Witamy w naszym wprowadzeniu do skryptów powłoki bash. W tym rozdziale poznasz rodzaje powłok dostępnych w systemie Linux oraz dowiesz się, dlaczego wybieramy powłokę bash. Dowiesz się, czym jest powłoka bash, jak napisać swój pierwszy skrypt i jak go uruchomić. Poza tym zobaczysz, jak skonfigurować popularne edytory tekstu w systemie Linux, takie jak vim i nano, do wprowadzania kodu źródłowego skryptów powłoki.

Podobnie jak w każdym innym języku skryptowym, jednym z najważniejszych elementów składowych używanych w skryptach powłoki bash są zmienne. Dzięki lekturze tego rozdziału dowiesz się, jak deklarować zmienne różnych typów, takich jak liczby całkowite, łańcuchy tekstu czy tablice, a także jak wyeksportować te zmienne i rozszerzyć ich zasięg poza bieżący proces.

Na koniec posiądziesz wiedzę o tym, jak można ułatwić sobie debugowanie kodu za pomocą pakietu Visual Studio Code.

W tym rozdziale omówimy następujące zagadnienia:

- rodzaje powłok systemu Linux;
- czym są skrypty powłoki bash?
- hierarchia poleceń powłoki bash;
- przygotowywanie edytorów tekstu do pisania skryptów;
- tworzenie i uruchamianie skryptów;
- deklarowanie zmiennych;

- zasięgi zmiennych;
- zastępowanie poleceń;
- debugowanie skryptów.

Wymagania techniczne

Do pracy z tą książką będziesz potrzebował działającego systemu Linux. Nie ma znaczenia, z której dystrybucji korzystasz, ponieważ wszystkie dystrybucje Linuksa są obecnie dostarczane z powłoką bash.

Pobierz i zainstaluj bezpłatny pakiet Visual Studio Code firmy Microsoft. Możesz go pobrać stąd: *https://code.visualstudio.com/*.

Pakietu Visual Studio Code możesz używać do tworzenia skryptów zamiast edytorów vim i nano; wybór należy do Ciebie.

My decydujemy się na korzystanie z edytora Visual Studio Code, ponieważ posiada całe mnóstwo przydatnych funkcji, takich jak uzupełnianie kodu, debugowanie i wiele innych.

Zainstaluj pakiet `bashdb`. Jest on wymagany dla wtyczki pozwalającej na debugowanie skryptów powłoki bash. Jeśli używasz dystrybucji opartej na systemie Red Hat, możesz zainstalować ten pakiet w następujący sposób:

```
$ sudo yum install bashdb
```

Jeżeli korzystasz z dystrybucji opartej na systemie Debian, pakiet `bashdb` możesz zainstalować w sposób ukazany poniżej:

```
$ sudo apt-get install bashdb
```

Zainstaluj wtyczkę Bash Debug dla pakietu Visual Studio Code, którą możesz pobrać z tej strony: *https://marketplace.visualstudio.com/items?itemName=rogalmic.bash-debug*. Zainstalowana wtyczka będzie używana do debugowania skryptów powłoki bash.

Kody źródłowe przykładów omawianych w tym rozdziale możesz pobrać z serwera FTP wydawnictwa Helion, pod adresem: *ftp://ftp.helion.pl/przyklady/skrzz2.zip*.

Rodzaje powłok systemu Linux

Jak zapewne już wiesz, system Linux zbudowany jest z kilku głównych elementów składowych, takich jak jądro, powłoka czy graficzny interfejs użytkownika (np. Gnome, KDE itd.).

Powłoka tłumaczy Twoje polecenia i przesyła je do systemu. Większość współczesnych dystrybucji systemu Linux jest dostarczana z wieloma powłokami.

Każda powłoka ma swój zestaw możliwości; niektóre powłoki systemu Linux są dziś bardzo popularne. Największą popularnością wśród programistów cieszą się te z poniższej listy:

- **Powłoka sh** — jest nazywana powłoką Bourne'a (*Bourne shell*), ponieważ opracował ją Stephen Bourne, a dokonał tego w latach 70. ubiegłego wieku w laboratoriach firmy AT&T. Jest to bardzo popularna powłoka oferująca wiele funkcji.
- **Powłoka bash** — nazwa powłoki jest akronimem angielskiej nazwy *Bourne Again Shell* (z ang. „jeszcze jedna powłoka Bourne'a"). Powłoka bash jest kompatybilna ze skryptami powłoki sh, dzięki czemu możesz w niej uruchamiać skrypty sh bez konieczności ich modyfikowania. W naszej książce będziemy korzystać właśnie z tej powłoki.
- **Powłoka ksh** — nazywa się ją również powłoką korn (*korn shell*). Jest kompatybilna z powłokami sh i bash. W porównaniu z powłoką bash powłoka ksh posiada kilka dodatkowych, ciekawych ulepszeń.
- **Powłoki csh i tcsh** — system Linux został napisany w języku C, co zainspirowało deweloperów z Berkeley University do opracowania powłoki, w której składnia poleceń jest podobna do składni poleceń języka C. Powłoka tcsh dodaje kilka drobnych usprawnień do bazowej powłoki csh.

Teraz znasz już rodzaje powłok i wiesz, że będziemy używać powłoki bash, więc nadszedł czas, aby powiedzieć kilka słów na temat skryptów tej powłoki.

Czym są skrypty powłoki bash?

Koncepcja skryptów powłoki bash polega na wykonywaniu sekwencji wielu poleceń w celu zautomatyzowania określonego zadania.

Jak pewnie wiesz, w wierszu poleceń powłoki możesz uruchamiać wiele z nich, oddzielając je od siebie średnikami (`;`), na przykład:

```
ls; pwd
```

Możemy powiedzieć, że powyższy wiersz poleceń to miniskrypt powłoki bash.

Najpierw zostaje wykonane pierwsze polecenie, a następnie uruchomione zostaje drugie polecenie.

Każde słowo kluczowe, które wpisujesz w skryptach powłoki bash, jest w rzeczywistości programem systemu Linux, nawet jeżeli jest to instrukcja `if`, `else` czy pętla `while`. Wszystkie instrukcje są wbudowanymi poleceniami powłoki lub plikami wykonywanymi (programami) tego systemu.

W pewnym sensie możemy powiedzieć, że powłoka jest klejem, który scala te polecenia.

Hierarchia poleceń powłoki bash

Kiedy siedzisz wygodnie przed monitorem i pracujesz z powłoką bash, która niecierpliwie czeka na wpisanie polecenia, możesz odnieść wrażenie, że jest to proste zadanie, które sprowadza się do wpisania nazwy polecenia i naciśnięcia klawisza *Enter*. W praktyce jednak niemal nic nigdy nie jest tak proste, jak to sobie wyobrażamy.

Typy poleceń

Jeżeli na przykład wpiszemy polecenie `ls`, które wyświetla listę plików, możemy rozsądnie myśleć, że uruchomiliśmy polecenie. Oczywiście może się tak zdarzyć, ale bardzo często w takiej sytuacji będziemy uruchamiać alias polecenia. Aliasy istnieją w pamięci powłoki jako skróty do określonych poleceń lub komend z listą opcji; aliasy są sprawdzane, zanim jeszcze powłoka zacznie poszukiwać danego pliku. Z pomocą może nam tutaj przyjść polecenie `type` powłoki bash, które wyświetla typ polecenia dla określonego słowa kluczowego wpisanego w wierszu poleceń. Istnieją następujące rodzaje poleceń:

- alias,
- funkcja,
- wewnętrzne (wbudowane) polecenie powłoki,
- słowo kluczowe,
- plik.

Powyższa lista jest również reprezentatywna dla kolejności, w jakiej wyszukiwany jest typ polecenia. Jak widać, pliku wykonywalnego `ls` powłoka będzie szukała dopiero na samym końcu.

Poniżej zamieszczamy prosty przykład użycia polecenia `type`:

```
$ type ls
ls jest aliasem do 'ls --color=auto'
```

Możemy rozszerzyć działanie tego polecenia tak, aby wyświetlić wszystkie dopasowania dla danego słowa kluczowego:

```
$ type -a ls
ls jest aliasem do 'ls --color=auto'
ls jest /bin/ls
```

Jeżeli potrzebna jest nam tylko nazwa typu polecenia, możemy skorzystać z opcji `-t`. Jest to przydatne, gdy musimy przetestować typ polecenia w skrypcie i potrzebujemy tylko nazwy typu. Użycie tej opcji wyłącza wyświetlanie wszelkich zbędnych informacji, a tym samym powoduje, że wyniki działania są bardziej czytelne. Przykładem zastosowania tej opcji może być następujące polecenie (zwróć uwagę na wynik jego działania):

```
$ type -t ls
alias
```

Wyniki działania są proste i przejrzyste, a zwykle jest to właśnie to, czego oczekuje Twój komputer lub skrypt.

Wbudowane polecenie `type` może również służyć do identyfikowania wbudowanych poleceń powłoki, takich jak `if` czy `case`. Poniższy przykład pokazuje sposób użycia polecenia `type` do identyfikacji wielu poleceń jednocześnie:

```
$ type ls quote pwd do id
```

Wyniki działania tego polecenia zostały przedstawione poniżej:

```
helion@helion:~$ type ls quote pwd do id
ls jest aliasem do 'ls --color=auto'
quote jest funkcją
quote ()
{
    local quoted=${1//\'/\'\\\'\'};
    printf "'%s'" "$quoted"
}
pwd jest wewnętrznym poleceniem powłoki
do jest słowem kluczowym powłoki
id jest /usr/bin/id
```

Zauważ, że jeżeli sprawdzany element jest funkcją, polecenie `type` wyświetla jego pełną definicję.

Zmienna środowiskowa PATH

System Linux poszukuje plików wykonywalnych w katalogach zdefiniowanych w zmiennej środowiskowej `PATH` tylko wtedy, gdy nie zostanie podana pełna lub względna ścieżka do programu. Na ogół bieżący katalog nie jest przeszukiwany, chyba że znajduje się w zmiennej `PATH`. W razie potrzeby możesz dołączyć katalog bieżący do zmiennej `PATH`, tak jak to zostało pokazane w przykładzie poniżej:

```
$ export PATH=$PATH:.
```

Wykonanie tego polecenia spowoduje dodanie bieżącego katalogu do zmiennej `PATH`; poszczególne ścieżki w zmiennej `PATH` są oddzielone od siebie dwukropkiem. Teraz zmieniliśmy zawartość zmiennej `PATH` tak, aby zawierała bieżący katalog roboczy, dzięki czemu za każdym razem, gdy zmienisz katalog, będziesz mógł łatwo uruchamiać przechowywane w nim skrypty.

Ogólnie rzecz biorąc, uporządkowanie skryptów w hierarchię katalogów jest bardzo dobrym rozwiązaniem. Na początek możesz w swoim katalogu domowym utworzyć podkatalog *bin* i zapisywać swoje skrypty w tym folderze. Dodanie ścieżki *$HOME/bin* do zmiennej `PATH` pozwoli Ci szybko wyszukiwać potrzebne skrypty po ich nazwie bez konieczności podawania ścieżki do pliku.

Polecenie przedstawione poniżej utworzy katalog *bin*, o ile taki katalog nie został utworzony wcześniej:

```
$ test -d $HOME/bin || mkdir $HOME/bin
```

Choć przeprowadzanie testu sprawdzającego, czy katalog *bin* już istnieje, nie jest w tym przypadku bezwzględnie konieczne, to jednak nasze przykładowe polecenie pokazuje, że tworzenie skryptów powłoki bash nie ogranicza się wyłącznie do zapisywania sekwencji poleceń w plikach, dzięki czemu równie dobrze możemy używać instrukcji warunkowych i innych konstrukcji programistycznych bezpośrednio z poziomu wiersza poleceń konsoli. Teraz już wiemy, że poprzednie polecenie zadziała niezależnie od tego, czy katalog *bin* już istnieje. Wykorzystanie zmiennej `$HOME` zapewnia poprawne działanie polecenia bez konieczności uwzględniania aktualnego położenia w systemie plików.

Podczas pracy z tą książką będziemy sukcesywnie dodawać różne skrypty do katalogu *$HOME/bin*, dzięki czemu będzie je można wykonywać niezależnie od tego, jaki będzie nasz bieżący katalog roboczy.

Przygotowywanie edytorów tekstu do pisania skryptów

W całej książce będziemy pracować z dystrybucją Linux Mint i będzie to obejmować również tworzenie i edycję skryptów. Rzecz jasna, możesz wybrać inny sposób edytowania swoich skryptów; na przykład jeżeli wolisz korzystać z edytora graficznego, pokażemy Ci niektóre przydatne ustawienia edytora gedit. W tym celu zrobimy małą wycieczkę do systemu Red Hat, w którym na potrzeby tego rozdziału przygotowaliśmy zrzuty okna edytora gedit.

Oprócz tego do edycji i debugowania naszych skryptów będziemy używać pakietu Visual Studio Code jako nowoczesnego edytora wyposażonego w wygodny, graficzny interfejs użytkownika (**GUI** — *Graphical User Interface*).

Aby ułatwić korzystanie z edytorów tekstu działających w konsoli z poziomu wiersza poleceń, możemy włączyć wybrane opcje i zapisać je w ukrytych plikach konfiguracyjnych. Gedit i inne edytory zaopatrzone w interfejs GUI zagwarantują nam podobną funkcjonalność.

Konfigurowanie edytora vim

Praca z edytorem tekstu jest po prostu koniecznością i częścią codziennego życia programisty. Skonfigurowanie wybranych opcji, ułatwiających korzystanie z edytora, pomoże nam zapewnić odpowiednią spójność podczas edytowania skryptów, co bez wątpienia będzie bardzo przydatne. Aby się o tym przekonać, ustawimy kilka użytecznych opcji w pliku konfiguracyjnym edytora vi lub vim, *$HOME/.vimrc*.

Oto zestawienie opcji, które będziemy ustawiać:

- `set showmode` — opcja ta powoduje wyświetlanie znacznika informującego, że pracujemy w trybie wstawiania.
- `set nohlsearch` — wyłącza podświetlanie słów, których szukaliśmy.
- `set autoindent` — w kodzie źródłowym skryptów często stosujemy wcięcia; włączenie tej opcji pozwala po naciśnięciu klawisza *Enter* automatycznie powrócić do poziomu ostatniego wcięcia, a nie na początek nowego wiersza.
- `set tabstop=4` — ustawia rozmiar tabulatora na cztery spacje.
- `set expandtab` — włącza automatyczne konwertowanie tabulatorów na spacje, co okazuje się przydatne, gdy plik jest przenoszony do innych systemów.
- `syntax on` — zwróć uwagę, że ta opcja nie używa polecenia `set`; powoduje włączenie podświetlania składni poleceń i słów kluczowych w skrypcie.

Po ustawieniu tych opcji plik *$HOME/.vimrc* powinien wyglądać mniej więcej tak:

```
set showmode
set nohlsearch
set autoindent
set tabstop=4
set expandtab
syntax on
```

Konfigurowanie edytora nano

Edytor nano zyskuje ostatnio na znaczeniu i jest domyślnym edytorem tekstu w wielu systemach. Nie podoba mi się w nim nawigacja w dokumencie, a w zasadzie jej brak, ale można go dostosować w taki sam sposób, jak robiliśmy to z edytorem vim. Tym razem jednak edytujemy plik *$HOME/.nanorc*. Po zakończeniu wprowadzania zmian plik konfiguracyjny powinien wyglądać mniej więcej tak:

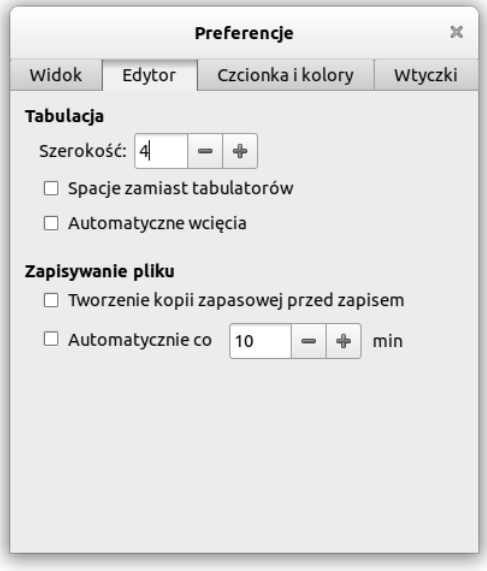
```
set autoindent
set tabsize 4
include /usr/share/nano/sh.nanorc
```

Ostatni wiersz włącza podświetlanie składni dla skryptów powłoki.

Konfigurowanie edytora gedit

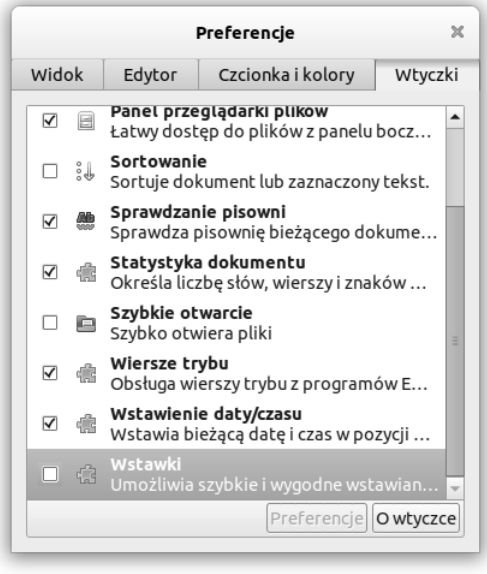
Edytory graficzne, takie jak gedit, mogą być konfigurowane za pomocą menu preferencji, dzięki czemu jest to bardzo proste.

Ustawienie tabulacji na 4 spacje i mechanizm zamiany tabulatorów na spacje możemy włączyć w oknie *Preferencje* na karcie *Edytor*, tak jak pokazano na poniższym zrzucie ekranu:



Pliki z kodami omawianych przykładów możesz pobrać z serwera FTP wydawnictwa Helion, pod adresem: *ftp://ftp.helion.pl/przyklady/skrzz2.zip*.

Kolejną bardzo użyteczną funkcję znajdziesz na karcie *Wtyczki* okna *Preferencje*, gdzie możemy włączyć wtyczkę *Wstawki* (ang. *Snippets*), która niezwykle ułatwia wstawianie gotowych fragmentów kodu:



W przykładach omawianych w dalszej części książki będziemy pracować z wierszem poleceń konsoli oraz edytorem vim, ale zachęcamy Cię do używania takiego edytora, z którym pracuje Ci się najlepiej. Tych kilka prostych wskazówek i zaleceń konfiguracyjnych, które przedstawiliśmy powyżej, stanowi dobry fundament do tworzenia skryptów i chociaż białe znaki, tabulatory i spacje w skryptach powłoki bash nie są znaczące, to jednak dobrze sformatowany plik kodu z odpowiednio ustawionymi odstępami jest znacznie łatwiejszy do przeglądania i analizowania. Co więcej, kiedy w dalszej części książki będziemy omawiać skrypty w języku Python, przekonasz się, że w niektórych językach programowania białe znaki mają ogromne znaczenie, dlatego lepiej będzie wyrobić w sobie dobre nawyki już na wczesnym etapie nauki programowania.

Tworzenie i wykonywanie skryptów

Po zakończeniu przygotowywania i konfigurowania edytorów możemy teraz szybko przejść do tworzenia i uruchamiania pierwszych skryptów. Jeżeli posiadasz już jakieś doświadczenia w pracy z systemem Linux i tworzeniem skryptów, lojalnie ostrzegamy, że zaczniemy od podstaw, ale będziemy również omawiać nieco bardziej zaawansowane zagadnienia, takie jak analizowanie parametrów pozycyjnych; oczywiście nic nie stoi na przeszkodzie, abyś pracował z tą książką we własnym tempie.

Witaj, świecie!

Jak zapewne doskonale zdajesz sobie z tego sprawę, każda dobra książka na temat pisania programów i skryptów tradycyjnie musi rozpoczynać się od skryptu „Witaj, świecie!", więc my również spróbujemy Cię tutaj nie zawieść. Zacznijmy zatem od utworzenia nowego skryptu, *$HOME/bin/hello1.sh*. Zawartość pliku powinna wyglądać tak, jak to zostało pokazane poniżej:

```
#!/bin/bash
echo "Witaj, świecie!"
exit 0
```

Mamy nadzieję, że ten prosty skrypt nie sprawił Ci żadnych kłopotów; w końcu to tylko trzy wiersze kodu. Gorąco zachęcamy Cię do uważnego analizowania i samodzielnego testowania wszystkich omawianych przykładów, co z pewnością pomoże Ci nabrać dodatkowego, praktycznego doświadczenia w pracy ze skryptami.

- `#!/bin/bash` — jest to z reguły pierwszy wiersz skryptu, nazywany często wierszem definicji interpretera lub po prostu *wierszem shebang*. Co ciekawe, wiersz shebang rozpoczyna się od znaku komentarza, ale mimo to jest analizowany i wykorzystywany przez system. W skryptach powłoki wiersze komentarza rozpoczynają się od znaku #. Wiersz shebang informuje system o tym, który interpreter powinien zostać zastosowany do uruchomienia i wykonania skryptu. Dla skryptów powłoki używamy zazwyczaj interpretera powłoki bash, a dla innych skryptów możemy, w zależności od potrzeb, użyć na przykład PHP lub

interpretera języka Perl. Jeżeli w skrypcie nie umieścimy wiersza shebang, polecenia będą uruchamiane za pomocą interpretera bieżącej powłoki; może to powodować problemy, gdy będziemy pracować z inną powłoką niż ta, dla której przeznaczony był uruchamiany skrypt.

- `echo "Witaj, świecie!"` — polecenie `echo` jest wbudowanym poleceniem powłoki, które umożliwia wysyłanie danych do standardowego wyjścia, `STDOUT`, którym domyślnie jest ekran. Ciąg znaków przeznaczony do wyświetlenia na ekranie jest ujęty w znaki cudzysłowu; więcej szczegółowych informacji na temat używania cudzysłowów, apostrofów i innych znaków znajdziesz w dalszej części książki.
- `exit 0` — polecenie `exit` jest wbudowanym poleceniem powłoki, które pozwala na przerwanie i zakończenie działania skryptu. Argumentem wywołania tego polecenia jest liczba całkowita reprezentująca kod zakończenia działania. Kod zakończenia o wartości innej niż 0 wskazuje na wystąpienie określonego błędu w działaniu skryptu.

Wykonywanie skryptu

Gotowy skrypt możemy zapisać w pliku na dysku, ale nawet jeżeli zapiszemy go w katalogu, którego ścieżka znajduje się w zmiennej środowiskowej `PATH`, to i tak nadal nie będzie jeszcze mógł być wykonywany jako samodzielny skrypt. Aby to zmienić, będziemy musieli nadać mu odpowiednie prawa pozwalające na wykonanie pliku. Dla potrzeb testowania możemy jednak uruchomić taki skrypt bezpośrednio za pomocą powłoki bash. W tym celu powinieneś wykonać polecenie pokazane poniżej:

```
$ bash $HOME/bin/hello1.sh
```

Po wykonaniu tego polecenia powinniśmy zostać nagrodzeni tekstem *Witaj, świecie!* wyświetlonym na ekranie. Nie jest to oczywiście rozwiązanie idealne, ponieważ docelowo chcielibyśmy ułatwić uruchamianie skryptu z dowolnej lokalizacji bez konieczności wpisywania pełnej ścieżki.

Aby to zrobić, musimy nadać plikowi skryptu odpowiednie uprawnienia do wykonywania, jak to pokazujemy w przykładzie poniżej:

```
$ chmod + x $HOME/bin/hello1.sh
```

Wykonawszy takie polecenie, powinieneś być w stanie uruchomić skrypt w prosty sposób. Przyjrzyj się poniższemu przykładowi:

```
helion@helios:~$ chmod +x $HOME/bin/hello1.sh
helion@helios:~$ hello1.sh
Witaj, świecie!
helion@helios:~$
```

Sprawdzanie statusu wyjścia

Nasz skrypt jest bardzo prosty, ale nie zmienia to faktu, że nadal musimy wiedzieć, jak używać kodów zakończenia skryptów i innych aplikacji. Sekwencja poleceń, którą stosowaliśmy podczas tworzenia katalogu *$HOME/bin*, jest dobrym przykładem tego, jak możemy posługiwać się kodem zakończenia:

```
$ polecenie1 || polecenie2
```

W powyższym przykładzie *polecenie2* jest wykonywane tylko wówczas, gdy próba wykonania *polecenia1* się nie powiedzie. Inaczej mówiąc, *polecenie2* zostanie wykonane wyłącznie wtedy, kiedy *polecenie1* zakończy działanie z kodem zakończenia innym niż 0.

Podobnie, w kolejnej sekwencji poleceń przedstawionej poniżej, *polecenie2* zostanie uruchomione tylko wtedy, gdy działanie *polecenia1* zakończy się powodzeniem i wygeneruje kod zakończenia o wartości 0:

```
$ polecenie1 && polecenie2
```

Aby odczytać wartość kodu zakończenia działania naszego skryptu, możemy sprawdzić wartość zmiennej $?, jak to zostało pokazane w poniższym przykładzie:

```
$ hello1.sh
$ echo $?
```

Oczekiwany wynik działania ostatniego polecenia to 0, ponieważ w ostatnim wierszu skryptu umieściliśmy odpowiednie polecenie `exit`, a w samym skrypcie praktycznie nie ma niczego, co mogłoby pójść źle i spowodować, że ostatni wiersz skryptu nie zostanie wykonany.

Zapewnienie unikalnej nazwy

Wiemy już, jak utworzyć i wykonać prosty skrypt, ale musimy jeszcze powiedzieć kilka słów na temat nadawania skryptom odpowiednich nazw. W przypadku naszego prostego skryptu nazwa *hello1.sh* będzie wystarczająco dobra i prawdopodobnie nie będzie kolidowała z innymi elementami systemu. Pamiętaj, że powinieneś unikać używania nazw, które mogą nie współgrać z istniejącymi aliasami, funkcjami, słowami kluczowymi i instrukcjami, a także nazw zainstalowanych już w systemie programów.

Dodanie przyrostka *.sh* do nazwy pliku nie gwarantuje, że taka nazwa będzie unikatowa, ale w systemie Linux, gdzie rozszerzenia nazw plików nie są używane, taki przyrostek staje się po prostu częścią nazwy pliku, co pomaga zapewnić unikalną nazwę skryptu. Dodatkowo niektóre edytory tekstu wykorzystują sufiksy do wybrania odpowiedniego pliku podświetlania składni. Jak zapewne pamiętasz, nieco wcześniej do edytora tekstu nano dołączyliśmy plik podświetlania składni *sh.nanorc*. Każdy z takich plików jest specyficzny dla określonego przyrostka nazwy i danego języka.

Wracając na chwilę do hierarchii poleceń powłoki bash, o której mówiliśmy na początku tego rozdziału, do określenia lokalizacji i typu pliku *hello1.sh* możemy użyć polecenia `type`:

```
$ type hello1.sh          # Wyświetla typ i ścieżkę polecenia
$ type -a hello1.sh       # Wyświetla wszystkie znalezione polecenia, jeżeli nazwa NIE jest
unikatowa
$ type -t hello1.sh       # Wyświetla typ polecenia
```

Wyniki działania tych poleceń zostały pokazane w kolejnym przykładzie poniżej:

```
helion@helios:~$ type hello1.sh
ścieżka do hello1.sh jest zapamiętana (/home/helion/bin/hello1.sh)
helion@helios:~$ type -a hello1.sh
hello1.sh jest /home/helion/bin/hello1.sh
helion@helios:~$ type -t hello1.sh
file
helion@helios:~$
```

Witaj, Gandalf!

Z całą pewnością w zdecydowanej większości przypadków w naszych skryptach będziemy potrzebować czegoś więcej niż tylko prostego wyświetlania statycznej wiadomości na ekranie. Oczywiście taka funkcjonalność również może być przydatna, ale możemy uczynić ten skrypt znacznie bardziej użytecznym, jeśli dodamy do niego pewną elastyczność.

W tym rozdziale przyjrzymy się parametrom pozycyjnym i argumentom wywołania skryptu, a w następnym rozdziale pokażemy, jak sprawić, by skrypt stał się bardziej interaktywny, i poprosimy użytkownika o wprowadzenie danych w czasie działania skryptu.

Uruchamianie skryptu z argumentami

Każdy skrypt możemy uruchomić z argumentami wywołania; w końcu jest to wolny świat, a system Linux promuje wolność i pozwala Ci z kodem zrobić wszystko. Jeżeli jednak skrypt nie używa żadnych argumentów wywołania, to wszelkie argumenty umieszczone w wierszu wywołania zostaną po prostu zignorowane. Poniżej pokazujemy przykład uruchomienia skryptu z pojedynczym argumentem wywołania:

```
$ hello1.sh Gandalf
```

Tak uruchomiony skrypt nadal będzie działał poprawnie i nie spowoduje błędu. Wyniki jego działania jednak się nie zmienią i na ekranie zobaczymy dobrze nam już znany komunikat *Witaj, świecie!*:

Identyfikator argumentu	Opis
$0	Nazwa samego skryptu, argument często używany przy wyświetlaniu składni oraz instrukcji użytkowania.
$1	Argument pozycyjny, reprezentujący pierwszy argument przekazany do skryptu.
${10}	Zapis stosowany w sytuacji, gdy do odwzorowania pozycji argumentu niezbędne jest użycie dwóch lub więcej cyfr. Nawiasy klamrowe służą do odseparowania nazwy zmiennej od innych treści. Przy mniejszej liczbie argumentów w tym miejscu oczekiwane są pojedyncze cyfry.
$#	Liczba argumentów wywołania skryptu; jest to szczególnie użyteczne, gdy musimy sprawdzić, ile argumentów wywołania jest potrzebnych do poprawnego wykonania skryptu.
$*	Reprezentuje wszystkie argumenty wywołania skryptu.

Aby skrypt mógł skorzystać z argumentu wywołania, musimy nieco zmienić kod skryptu. Żeby to zrobić, najpierw utwórz kopię naszego skryptu, nadaj jej nazwę *hello2.sh* i dodaj uprawnienia do wykonywania:

```
$ cp $HOME/bin/hello1.sh $HOME/bin/hello2.sh
$ chmod +x $HOME/bin/hello2.sh
```

Teraz musimy otworzyć plik *hello2.sh* do edycji i zmienić kod skryptu, tak aby skorzystać z argumentu przekazywanego w wierszu poleceń. Poniższy zrzut ekranu pokazuje najprostsze użycie argumentów wywołania, które pozwala nam teraz na wyświetlenie powitania dla dowolnej nazwy użytkownika:

```
#!/bin/bash
echo "Witaj, $1!"
exit 0
```

Uruchom teraz nową wersję skryptu, podając jako argument wywołania dowolną nazwę użytkownika, tak jak to zostało pokazane poniżej:

```
$ hello2.sh Gandalf
```

Wyniki działania skryptu powinny wyglądać tak: *Witaj, Gandalf!* Jeżeli w wierszu wywołania nie podamy żadnego argumentu, to zmienna go reprezentująca będzie pusta i w wynikach działania pojawi się tylko słowo *Witaj*. Przykładowe wyniki działania są następujące:

```
helion@helios:~$ hello2.sh Gandalf
Witaj, Gandalf!
helion@helios:~$
```

Jeśli dostosujemy skrypt do użycia zmiennej $*, wyświetlone zostaną wartości wszystkich argumentów wywołania. W naszym przypadku zobaczymy słowo *Witaj*, a w dalszej kolejności wartości wszystkich podanych argumentów wywołania. Zmodyfikuj skrypt tak, aby wiersz polecenia `echo` wyglądał w następujący sposób:

```
echo "Witaj, $*!"
```

Następnie uruchom skrypt z takimi argumentami:

```
$ hello2.sh Gandalf Frodo Aragorn Saruman
```

Wynik działania takiego polecenia pokazany został na poniższym zrzucie ekranu:

```
helion@helios:~$
helion@helios:~$ hello2.sh Gandalf Frodo Aragorn Saruman
Witaj, Gandalf Frodo Aragorn Saruman!
helion@helios:~$
```

Jeżeli chciałbyś wyświetlać komunikaty *Witaj, <imię>* dla każdego z użytkowników w osobnym wierszu, to będziesz musiał poczekać, aż dotrzemy do omawiania pętli `for`, która będzie dobrym rozwiązaniem dla takiego zadania.

Dlaczego odpowiednio dobrane cudzysłowy i apostrofy są takie ważne?

Do tej pory wszystkie ciągi znaków, które wyświetlaliśmy na ekranie za pomocą polecenia `echo`, ujmowaliśmy w znaki cudzysłowu.

W naszym pierwszym skrypcie (*hello1.sh*) nie miało znaczenia, czy używamy cudzysłowu czy apostrofów. Wyniki działania polecenia `echo "Witaj, świecie!"` będą dokładnie takie same jak wyniki polecenia `echo 'Witaj, świecie!'`.

W przypadku drugiego skryptu (*hello2.sh*) już tak jednak nie jest, dlatego dobre zrozumienie mechanizmów cytowania dostępnych w powłoce bash jest niezmiernie ważne.

Jak mogłeś się sam przekonać, używanie znaków cudzysłowu w wierszu polecenia `echo "Witaj, $1!"` spowodowało, że w wynikach działania pojawiał się komunikat odzwierciedlający wartość argumentu podanego w wierszu wywołania polecenia, na przykład *Witaj, Gandalf!*. Jeżeli jednak w tym samym wierszu zamiast cudzysłowu zastosujemy apostrofy, na ekranie pojawi się komunikat *Witaj, $1!*, co oznacza, że zamiast wartości zmiennej wyświetlana jest jej nazwa.

Ujmowanie ciągów znaków w apostrofy ma na celu ochronę znaków specjalnych, takich jak spacja między dwoma słowami, przed błędną interpretacją. Bez apostrofów powłoka traktuje spacje jako separatory argumentów wywołania polecenia. Umieszczenie ciągu znaków w apostrofach powoduje, że cały ciąg znaków jest traktowany jako literał, którego poszczególne znaki są po prostu tekstem i nie mają żadnego specjalnego znaczenia. Ma to jednak druzgocący wpływ na zmienne rozpoczynające się od znaku $, ponieważ w takiej sytuacji powłoka nie może rozwijać zmiennych i zamiast wartości zmiennych wyświetla w ciągach znaków ich nazwy.

W takiej sytuacji z pomocą przychodzą nam cudzysłowy. Użycie cudzysłowów powoduje ochronę wszystkich znaków z wyjątkiem $, co pozwala powłoce bash na rozwijanie zmiennych i zastępowanie ich nazw odpowiednimi wartościami.

Jeżeli kiedykolwiek będziesz musiał użyć literału $ w cytowanym ciągu zawierającym zmienne, które muszą zostać rozwinięte, możesz posłużyć się cudzysłowami i jednocześnie wyłączyć interpretację pożądanego znaku $ za pomocą lewego ukośnika (\), spełniającego rolę znaku ucieczki. Na przykład polecenie `echo "$USER zarobił \$4"` spowoduje wyświetlenie komunikatu *Gandalf zarobił $4* (zakładając oczywiście, że aktualnie zalogowanym użytkownikiem jest *Gandalf*).

Poniżej przedstawiamy kilka wersji tego samego polecenia, wykorzystujących różne mechanizmy cytowania. Spróbuj samodzielnie wykonać takie polecenia i przeanalizuj otrzymane wyniki.

```
$ echo "$USER zarobił $4"
$ echo '$USER zarobił $4'
$ echo "$USER zarobił \$4"
```

Wyniki działania powyższych poleceń zostały pokazane poniżej:

```
helion@helios:~$ echo "$USER zarobił $4"
helion zarobił
helion@helios:~$ echo '$USER zarobił $4'
$USER zarobił $4
helion@helios:~$ echo "$USER zarobił \$4"
helion zarobił $4
helion@helios:~$
```

Wyświetlanie nazwy skryptu

Zmienna $0 reprezentuje nazwę skryptu i jest często stosowana przy wyświetlaniu składni oraz instrukcji użytkowania skryptu. Ponieważ nie omawialiśmy jeszcze instrukcji warunkowych, spróbujemy wyświetlić nazwę skryptu bezpośrednio nad naszym komunikatem powitalnym.

Zmodyfikuj kod skryptu *$HOME/bin/hello2.sh* tak, aby wyglądał jak w przykładzie przedstawionym poniżej:

```
#!/bin/bash
echo "Używasz skryptu $0"
echo "Witaj, $*!"
exit 0
```

Wyniki działania skryptu prezentują się następująco:

```
helion@helios:~$ hello2.sh Gandalf
Używasz skryptu /home/helion/bin/hello2.sh
Witaj, Gandalf!
helion@helios:~$
```

Jeśli nie chcemy wyświetlać całej ścieżki i zamiast tego chcemy tylko pokazać na ekranie nazwę skryptu, możemy użyć polecenia `basename`, które wyodrębnia nazwę ze ścieżki. Zmodyfikuj skrypt w taki sposób, by jego drugi wiersz wyglądał następująco:

```
echo "Używasz skryptu $(basename $0)"
```

Składnia `$(....)` pozwala na pobranie wyniku działania polecenia znajdującego się wewnątrz nawiasów. W naszym przypadku najpierw wykonywane jest polecenie `basename $0`, a jego wynik działania jest przekazywany do nienazwanej zmiennej reprezentowanej przez `$`.

Wyniki działania zmodyfikowanego skryptu zostały pokazane poniżej:

```
helion@helios:~$ hello2.sh Gandalf
Używasz skryptu hello2.sh
Witaj, Gandalf
helion@helios:~$
```

Takie same wyniki można również osiągnąć za pomocą grawisów („odwrotnych apostrofów"). Taki zapis jest co prawda mniej czytelny, ale warto o nim wspomnieć, ponieważ możesz się z nim spotkać podczas analizowania i modyfikowania skryptów napisanych przez innych użytkowników. Alternatywna składnia z wykorzystaniem grawisów została pokazana w poniższym przykładzie:

```
echo "Używasz skryptu `basename $0`"
```

Pamiętaj, że użyte tutaj znaki to grawisy, a NIE apostrofy proste. Na standardowych klawiaturach grawisy znajdują się w lewym górnym rogu obok klawisza z cyfrą 1.

Deklarowanie zmiennych

Podobnie jak praktycznie w każdym języku programowania, w skryptach powłoki bash można deklarować zmienne, a zatem warto powiedzieć kilka słów o tym, czym są zmienne i jakie są zalety ich używania.

Zmienna odgrywa rolę swego rodzaju pudełka, w którym możesz przechowywać różne wartości do późniejszego wykorzystania w kodzie.

Istnieją dwa rodzaje zmiennych, które możesz zadeklarować w skrypcie:

- zmienne definiowane przez użytkownika,
- zmienne środowiskowe.

Zmienne definiowane przez użytkownika

Aby zadeklarować zmienną użytkownika, wystarczy wpisać żądaną nazwę i za pomocą znaku równości (`=`) przypisać jej odpowiednią wartość.

Sprawdź taki przykład:

```
#!/bin/bash
name="Gandalf"
age=35
total=16.5
```

```
echo $name   # wyświetla ciąg znaków Gandalf
echo $age    # wyświetla liczbę 35
echo $total  # wyświetla liczbę 16.5
```

Jak widać w przykładzie, aby wyświetlić na ekranie wartość zmiennej, powinieneś umieścić przed nią znak dolara ($).

Zauważ, że ani pomiędzy nazwą zmiennej a znakiem równości, ani między znakiem równości a wartością zmiennej nie ma żadnych spacji.

Jeżeli przez pomyłkę wpiszesz taką spację, powłoka potraktuje zmienną tak, jakby była poleceniem, a ponieważ zazwyczaj nie ma polecenia o takiej nazwie, powłoka wyświetli błąd.

Poniżej zamieszczamy kilka przykładów niepoprawnych deklaracji zmiennych:

```
# Nie deklaruj zmiennych w przedstawiony poniżej sposób:
name =  "Gandalf"
age   =35
total= 16.5
```

Kolejnym przydatnym typem zmiennych definiowanych przez użytkownika są tablice. Tablica może przechowywać wiele wartości, zatem jeżeli masz dziesiątki wartości, których chcesz użyć w swoim skrypcie, to zamiast zapełniać kod zmiennymi, powinieneś po prostu skorzystać z tablicy.

Aby zadeklarować tablicę, należy umieścić jej elementy w nawiasach, tak jak to zostało pokazane poniżej:

```
#!/bin/bash
myarr=(jeden dwa trzy cztery pięć)
```

Żeby odwołać się do wybranego elementu tablicy, powinieneś użyć odpowiedniego numeru indeksu, tak jak to zostało pokazane w kolejnym przykładzie:

```
#!/bin/bash
myarr=(jeden dwa trzy cztery pięć)
echo ${myarr[1]} # wyświetla ciąg znaków "dwa", będący drugim elementem tablicy
```

Pamiętaj, że numerowanie elementów tablic zawsze rozpoczyna się od zera.

By wyświetlić wszystkie elementy tablicy, zamiast numeru elementu użyj symbolu gwiazdki:

```
#!/bin/bash
myarr=(jeden dwa trzy cztery pięć)
echo ${myarr[*]}
```

Aby usunąć z tablicy wybrany element, powinieneś użyć polecenia `unset`:

```
#!/bin/bash
myarr=(jeden dwa trzy cztery pięć)
unset myarr[1] # To polecenie usuwa drugi element tablicy
unset myarr    # To polecenie usuwa wszystkie elementy tablicy
```

Zmienne środowiskowe

Do tej pory posługiwaliśmy się zmiennymi, których wcześniej nie definiowaliśmy, i były to: `$BASH_VERSION`, `$HOME`, `$PATH` oraz `$USER`. Być może zadajesz sobie pytanie: skąd one się wzięły, skoro nie zostały przez nas zadeklarowane?

Są to tzw. zmienne środowiskowe, definiowane przez samą powłokę.

Istnieje wiele zmiennych środowiskowych. Jeżeli chcesz je wyświetlić, możesz użyć polecenia `printenv`.

Aby wyświetlić wybraną zmienną środowiskową, powinieneś podać jej nazwę jako argument wywołania polecenia `printenv`:

```
$ printenv HOME
```

W skryptach powłoki bash możemy używać dowolnej z tych zmiennych.

Zwróć uwagę, że nazwy wszystkich zmiennych środowiskowych są pisane wielkimi literami, wobec czego w nazwach swoich zmiennych powinieneś stosować małe litery, aby ułatwić ich odróżnienie od zmiennych środowiskowych. Nie jest to co prawda wymagane, ale mocno zalecane.

Zasięg zmiennych

Po zadeklarowaniu zmiennej będzie ona dostępna do użycia w całym skrypcie powłoki bash.

Załóżmy teraz następujący scenariusz: podzieliłeś swój skrypt na dwa pliki i wywołujesz jeden z nich z poziomu drugiego, na przykład:

```
# Pierwszy skrypt
#!/bin/bash
name="Gandalf"
./script2.sh      # Wywołanie drugiego skryptu
```

Drugi skrypt wygląda następująco:

```
# Drugi skrypt: skrypt2.sh
#!/bin/bash
echo $name
```

Załóżmy, że chcesz wykorzystać zmienną `name` w drugim skrypcie. Jeżeli jednak spróbujesz wyświetlić jej wartość, nic się nie pojawi — dzieje się tak dlatego, że domyślnie zasięg zmiennej jest ograniczony tylko do procesu, który ją tworzy.

Aby użyć zmiennej `name` w innych skryptach, możesz wyeksportować ją za pomocą polecenia `export`.

Odpowiednio zmodyfikowany skrypt będzie zatem wyglądał następująco:

```
# Pierwszy skrypt
#!/bin/bash
name="Gandalf"
export name       # Zmienna będzie dostępna dla innych procesów
./skrypt2.sh
```

Teraz, kiedy uruchomisz nasz program, pierwszy skrypt utworzy i wyeksportuje zmienną `name`, a następnie wywoła drugi skrypt, który wyświetli na ekranie wartość zmiennej `name` zadeklarowanej w pierwszym skrypcie.

Pamiętaj, że drugi skrypt, *skrypt2.sh*, tworzy tylko kopię zmiennej i nigdy nie modyfikuje jej oryginału.

By się o tym przekonać, spróbuj w drugim skrypcie zmienić wartość tej zmiennej, a potem spróbuj uzyskać dostęp do nowej wartości zmiennej z poziomu pierwszego skryptu:

```
# Pierwszy skrypt
#!/bin/bash
name="Gandalf"
export name
./skrypt2.sh
echo $name
```

Drugi skrypt powinien wyglądać tak:

```
# Drugi skrypt: skrypt2.sh
#!/bin/bash
name="Inne imię"
echo $name
```

Jeżeli teraz uruchomisz nasz program, na ekranie wyświetlona zostanie zmodyfikowana wartość zmiennej `name` z drugiego skryptu, a następnie wyświetlona zostanie oryginalna wartość z pierwszego skryptu — pierwotna zmienna pozostaje zatem bez zmian.

Podstawianie wyników działania poleceń

Pokazaliśmy już, jak możesz deklarować zmienne, które przechowują liczby całkowite, łańcuchy znaków, tablice albo liczby zmiennoprzecinkowe. Ale to nie wszystko.

Podstawianie poleceń oznacza możliwość przechowywania wyników działania danego polecenia w zmiennej.

Jak być może już wiesz, polecenie `pwd` wyświetla nazwę bieżącego katalogu roboczego. Zobaczymy więc, jak zapisać wyniki działania tego polecenia w zmiennej.

Istnieją dwa sposoby podstawiania wyników działania poleceń do zmiennej:

- używanie grawisów (`);
- używanie zapisu ze znakiem dolara, takiego jak : `$( )`.

Korzystając z pierwszej metody, po prostu umieszczamy polecenie między dwoma grawisami:

```
#!/bin/bash
cur_dir=`pwd`
echo $cur_dir
```

Zastosowanie drugiej metody wygląda następująco:

```
#!/bin/bash
cur_dir=$(pwd)
echo $cur_dir
```

Wyniki działania poszczególnych poleceń mogą być dalej przetwarzane i na ich podstawie podejmowane mogą być różne działania.

Debugowanie skryptów

Jeżeli skrypty są proste, takie jak omawialiśmy do tej pory, to po ich uruchomieniu zazwyczaj niewiele może pójść nie tak, jak trzeba, lub wymagać debugowania. W miarę rozbudowywania skryptu i wprowadzania wielu złożonych instrukcji warunkowych analizowanie przebiegu jego działania może wymagać zastosowania debugowania.

Powłoka bash posiada dwie opcje ułatwiające debugowanie skryptów, `-v` i `-x`.

Jeżeli chcesz zobaczyć rozbudowane wyniki działania skryptu oraz wyświetlić szczegółowe informacje o kolejnych wierszach wykonywanego kodu, możesz użyć opcji `-v`. Możesz to zrobić w wierszu shebang skryptu, ale często znacznie łatwiej jest uruchomić skrypt bezpośrednio z poziomu wiersza poleceń za pomocą powłoki bash:

```
$ bash -v $HOME/bin/hello2.sh Gandalf
```

Jest to szczególnie przydatne w naszym przykładzie, ponieważ możemy zobaczyć, jak przetwarzane są poszczególne elementy osadzonej komendy `basename` — w pierwszym kroku usuwane są znaki cudzysłowu, a następnie nawiasy. Wyniki działania tego polecenia zostały pokazane poniżej:

```
helion@helios:~$ bash -v $HOME/bin/hello2.sh Gandalf
#!/bin/bash
echo "Używasz skryptu $(basename $0)"
basename $0)"
basename $0)
basename $0
Używasz skryptu hello2.sh
echo "Witaj, $*"
Witaj, Gandalf
exit 0
helion@helios:~$
```

Opcja -x, która wyświetla polecenia w czasie ich wykonywania, jest używana znacznie częściej, ponieważ pozwala użytkownikowi poznać gałęzie decyzyjne wybrane przez skrypt. Poniżej przedstawiamy wyniki działania naszego skryptu uruchomionego z tą opcją:

```
$ bash -x $HOME/bin/hello2.sh fred
```

I znów możemy zauważyć, że najpierw wykonywana jest komenda basename, ale nie widzimy żadnych szczegółowych kroków związanych z jej uruchomieniem. Przykładowe wyniki działania tego polecenia zostały pokazane poniżej:

```
helion@helios:~$ bash -x $HOME/bin/hello2.sh Gandalf
++ basename /home/helion/bin/hello2.sh
+ echo 'Używasz skryptu hello2.sh'
Używasz skryptu hello2.sh
+ echo 'Witaj, Gandalf'
Witaj, Gandalf
+ exit 0
helion@helios:~$
```

Przedstawiona metoda może jednak sprawiać pewne kłopoty, zwłaszcza początkującym użytkownikom lub użytkownikom, którzy mają co prawda doświadczenie w programowaniu, ale do tej pory korzystali z pakietów pozwalających na debugowanie kodu w środowisku graficznym.

Innym, znacznie bardziej nowoczesnym sposobem debugowania skryptów powłoki jest użycie programu Visual Studio Code (VSC).

Dla VSC istnieje wtyczka o nazwie bash debug, która umożliwia wizualne debugowanie skryptów bash, tak jak ma to miejsce w przypadku wielu innych języków programowania.

Wtyczka pozwala na wykonywanie kodu krok po kroku, funkcja po funkcji, dodawanie pułapek i realizację wielu innych operacji związanych z debugowaniem, do których jesteś przyzwyczajony.

Po zainstalowaniu wtyczki przejdź do menu *File* i otwórz folder shell-scripts. Następnie możesz skonfigurować proces debugowania, naciskając kombinację klawiszy *Ctrl+Shift+P* i wpisując takie oto polecenie:

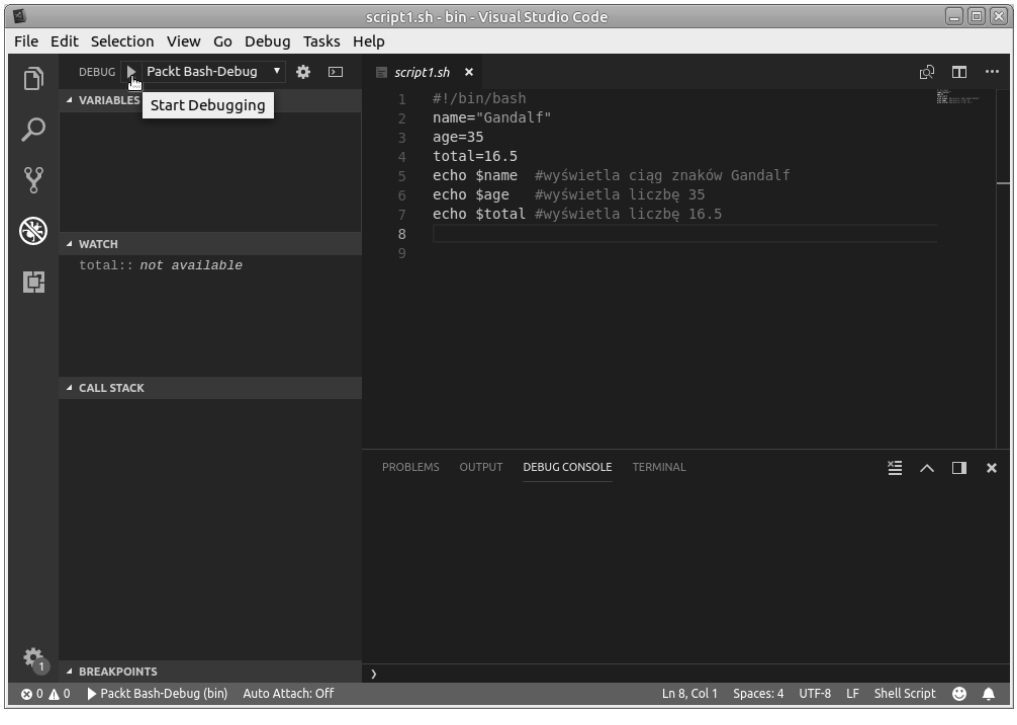
```
Debug:open launch.json
```

Spowoduje to otwarcie nowego, pustego pliku, w którym powinieneś wpisać następującą konfigurację:

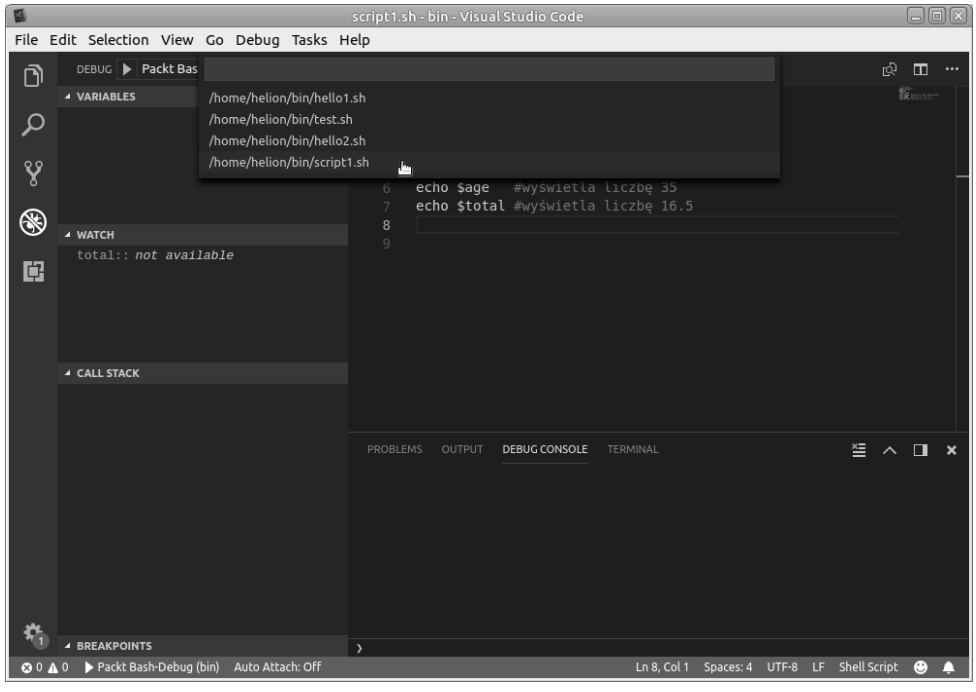
```
{
   "version": "0.2.0",
   "configurations": [
      {
         "name": "Packt Bash-Debug",
         "type": "bashdb",
```

```
            "request": "launch",
            "scriptPath": "${command:SelectScriptName}",
            "commandLineArguments": "",
            "linux": {
               "bashPath": "bash"
            },
            "osx": {
               "bashPath": "bash"
            }
         }
      ]
   }
```

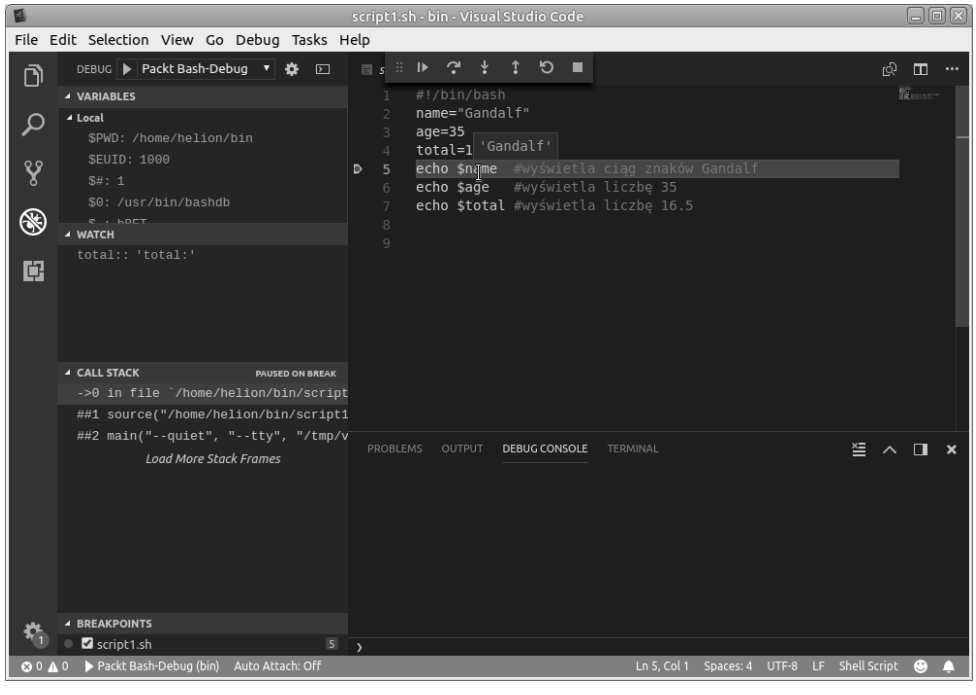
Po zapisaniu tego pliku utworzona zostanie konfiguracja debugowania o nazwie `Packt Bash-Debug`, tak jak to zostało pokazane na poniższym rysunku:



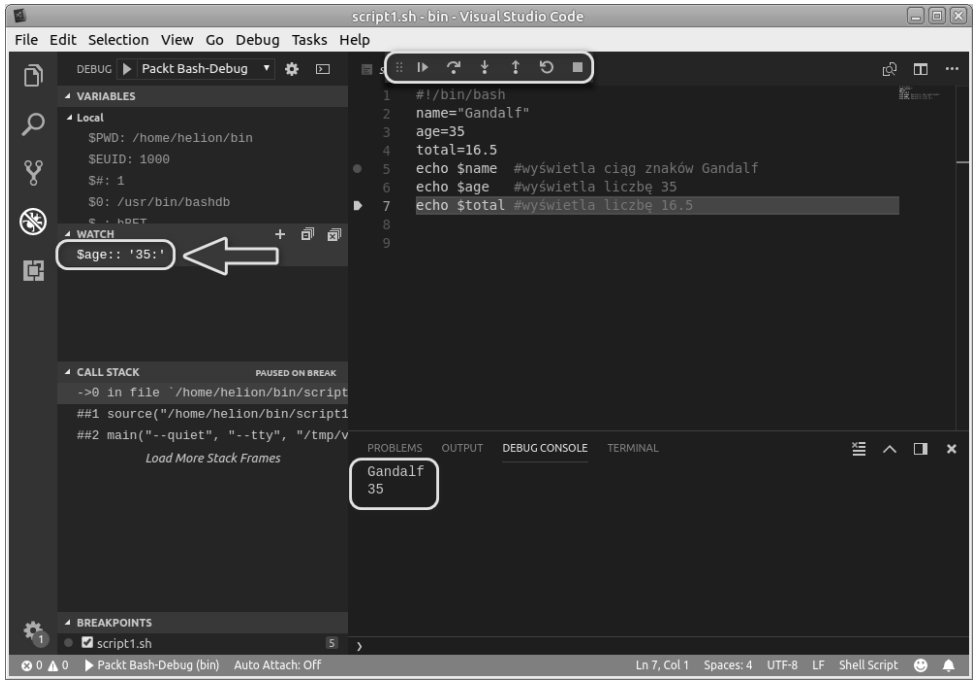
Teraz rozpocznij debugowanie, naciskając klawisz *F5* lub wybierając z głównego menu polecenie *Debug/Start Debugging*; na ekranie zostanie wyświetlona lista dostępnych plików *.sh*:



Wybierz z listy plik, który chcesz debugować, i w dowolnym wierszu kodu ustaw punkt przerwania (pułapkę), tak jak to zostało pokazane na rysunku poniżej:



Wykonując teraz kolejne wiersze kodu, możesz dodawać wybrane zmienne do listy obserwowanych (*watch list*), dzięki czemu będziesz mógł przeglądać ich wartości:



> Pamiętaj, że Twój skrypt **MUSI** rozpoczynać się od wiersza shebang, `#!/bin/bash`.

Od tej chwili możesz cieszyć się możliwością wygodnego debugowania skryptów w środowisku graficznym. Pomyślnego kodowania!

Podsumowanie

Na tym zakończymy omawianie zagadnień w tym rozdziale i mamy nadzieję, że jego zawartość okazała się dla Ciebie przydatna. Rozdział ten stanowi swego rodzaju wstęp do tworzenia skryptów powłoki bash, zwłaszcza dla tych użytkowników, którzy dopiero rozpoczynają swoją przygodę z programowaniem.

Rozpoczęliśmy od upewnienia się, że bash jest bezpieczną powłoką i posiada wiele wbudowanych, przydatnych funkcji i poleceń. Następnie omówiliśmy hierarchię wykonywania poleceń powłoki, gdzie dowiedziałeś się, że aliasy i funkcje są sprawdzane przed wbudowanymi poleceniami powłoki, poznałeś dobre praktyki w nadawaniu nazw skryptom i dowiedziałeś się, jak skonfigurować zmienną środowiskową `$PATH`, tak aby zawierała ścieżkę do katalogu, w którym przechowujesz swoje skrypty.

Następnie omówiliśmy rodzaje powłok dostępnych w systemie Linux i powiedzieliśmy kilka słów o tym, czym są skrypty powłoki bash.

Zaraz po tym rozpoczęliśmy pisanie prostych skryptów ze statyczną zawartością i pokazaliśmy, jak łatwo można zwiększyć elastyczność skryptu za pomocą argumentów wywołania. Ponadto dowiedziałeś się, jak za sprawą zmiennej `$?` sprawdzić wartość kodu zakończenia działania skryptu oraz w jaki sposób w wierszu poleceń konsoli można tworzyć złożone sekwencje poleceń z wykorzystaniem operatorów `||` i `&&`, pozwalających na warunkowe wykonywanie kolejnych poleceń sekwencji w zależności od statusu wykonania poprzedniego polecenia.

Następnie pokazaliśmy, jak deklarować zmienne i jak używać zmiennych środowiskowych. Omówiliśmy podstawowe zagadnienia związane z zasięgiem zmiennych i wyjaśniliśmy, jak eksportować zmienne do innego procesu.

Z tego rozdziału dowiedziałeś się również, jak przechowywać wyniki działania poleceń w zmiennych. Taki proces nazywamy podstawianiem wyników działania poleceń.

Na koniec omówiliśmy możliwości debugowania skryptów za pomocą powłoki bash oraz programu Visual Studio Code. Oczywiście w przypadku prostych skryptów debugowanie nie jest konieczne, ale w miarę wzrostu stopnia złożoności skryptu staje się coraz bardziej przydatne.

W następnym rozdziale opiszemy, jak tworzyć interaktywne skrypty, które po uruchomieniu potrafią pobierać dane wejściowe od użytkownika.

Pytania

1. Na czym polega problem z następującym kodem skryptu? Jak możemy to naprawić?

```
#!/bin/bash
var1 ="Witamy w skryptach powłoki bash ..."
echo $var1
```

2. Jaki będzie wynik działania poniższego skryptu?

```
#!/bin/bash
arr1=(Sobota Niedziela Poniedziałek Wtorek Środa)
echo ${arr1[3]}
```

3. Na czym polega problem z następującym kodem skryptu? Jak możemy to naprawić?

```
#!/bin/bash
files = 'ls -la'
echo $files
```

4. Jaka będzie wartość zmiennych `b` i `c` w poniższym skrypcie?

```
#!/bin/bash
a=15
b=20
c=a
b=c
```

Co dalej?

Jeżeli zainteresowały Cię zagadnienia omawiane w tym rozdziale, więcej szczegółowych informacji znajdziesz na następujących stronach internetowych:

- *http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO-5.html*,
- *http://tldp.org/LDP/abs/html/varassignment.html*,
- *http://tldp.org/LDP/abs/html/declareref.html*.

Opanuj sztukę pisania doskonałych skryptów powłoki!

Mimo że nowe wydania dystrybucji Linuksa są coraz łatwiejsze w obsłudze, a ważniejsze czynności administracyjne mogą być wykonywane za pomocą intuicyjnego interfejsu graficznego, wciąż nie można się obejść bez powłoki Bourne'a, znanej jako bash. Dobrze napisany skrypt powłoki pozwala na automatyzację nudnych obowiązków, umożliwia monitorowanie stanu systemu, optymalizację jego wydajności czy dostosowanie go do potrzeb. Warto też wypróbować ciekawą alternatywę dla tradycyjnych skryptów powłoki bash, czyli kod Pythona.

Dzięki tej książce nauczysz się wszystkiego, co jest potrzebne do pisania profesjonalnych skryptów powłoki. Dowiesz się, czym są powłoki systemu Linux, dlaczego tak ważna jest powłoka bash i w jaki sposób edytuje się skrypty. Nauczysz się pracy na zmiennych, debugowania kodu i tworzenia skryptów interaktywnych. Będziesz korzystać z instrukcji warunkowych i pętli, a także z edytora vim, pakietu Visual Studio Code oraz edytora strumieniowego sed. Zapoznasz się z zasadami pisania funkcji, dzięki którym będziesz mógł wielokrotnie używać uniwersalnych fragmentów kodu. Ponadto zdobędziesz umiejętność przetwarzania danych tekstowych zarówno za pomocą polecenia AWK, jak i wyrażeń regularnych. Na koniec przekonasz się, jak ciekawą alternatywą dla skryptów powłoki bash jest kod napisany w Pythonie!

W tej książce między innymi:

- wyczerpujące wprowadzenie do tworzenia i debugowania skryptów powłoki
- składnia alternatywna i operacje arytmetyczne
- praca z blokami kodu i korzystanie z funkcji
- automatyzacja tworzenia hostów wirtualnych
- zaawansowane korzystanie z polecenia AWK
- skrypty do analizy plików dziennika i tworzenia raportów

Mokhtar Ebrahim od 2010 roku administruje systemami Linux. Jego pasją jest tworzenie skryptów powłoki bash i programów w Pythonie, automatyzujących wiele codziennych zadań. Pisze artykuły techniczne dla serwisu Like Geeks.
Andrew Mallett jest autorem książek i właścicielem The Urban Penguin. Tworzy profesjonalne oprogramowanie i prowadzi szkolenia z zakresu systemów Linux. Jest wielkim fanem wiersza poleceń i uważa skrypty za nieodzowne narzędzie każdego admina.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
INFORMATYKA W NAJLEPSZYM WYDANIU

ISBN 978-83-283-5070-0
9 788328 350700
Cena: 59,00 zł
Packt