
Warsztat
Programisty

Warsztat Programisty – materiały do zajęć

Łukasz Kuszner, Radosław Ziemann

Uniwersytet Gdański
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki
Zakład Optymalizacji Kombinatorycznej

28 października 2025

Spis treści

Rozdział 1

Wprowadzenie

Podręcznik został napisany dla studentów informatyki Uniwersytetu Gdańskiego na potrzeby przedmiotu *Warsztat programisty*. W ramach tego przedmiotu skupiamy się na tak zwanym *przetwarzaniu wsadowym*[1] (ang. *batch processing*), czyli wykonywaniu *skryptów* i *programów* bez interakcji z użytkownikiem, oraz narzędziach użytecznych w tworzeniu tak działających programów.

Powłoka

Do pisania i uruchamiania skryptów posłuży nam *interpreter* Bash (ang. *Bourne Again SHell*), inaczej *powłoka* (ang. *shell*) systemu Linux. Tak więc nasze *skrypty* będą programami napisanymi w języku powłoki Bash.

Powłoka, oprócz wykonywania skryptów, umożliwia też pracę interaktywną – manualne wprowadzanie poleceń. Program, który otwiera okno pozwalające nam na komunikację z powłoką nazywamy *terminalem*. Oprócz Basha istnieją też inne powłoki, np: `sh`, `csh`, `tcsh`, `ksh`, `zsh` – nie będziemy o nich tutaj mówić.

Język C i kompilator

Język C to język *kompilowany* w przeciwieństwie do interpretowanego Basha. Polecenia programu napisanego w C wymagają innego programu – *kompilatora* do przetworzenia kodu programu na *język maszyny* lub inaczej mówiąc *kod wykonywalny*.

Kompilacja, a więc tłumaczenie przez kompilator kodu programu na kod wykonywalny, jest na ogół procesem wieloetapowym. Na nasze potrzeby rozróżnimy trzy główne etapy: etap wstępny, gdy wykonywane są makra preprocesora (ang. *preprocessing*), kompilacja pojedynczych plików i *łączenie* lub inaczej *konsolidacja* (ang. *linking*) – ostatni etap, w którym efekt działania w poprzednich etapach jest łączony w docelowe pliki wykonywalne.

Narzędzia budowania projektu

W ramach kursu będziemy ćwiczyć z małymi projektami, ale docelowo, jako programiści, pracujemy ze złożonymi projektami składającymi się z wielu *jednostek kompilacji*. Zapoznamy się z programami `make` i `cmake` pozwalającymi na automatyczne zarządzanie procesem kompilacji.

[1] Istotne pojęcia używane po raz pierwszy są wyróżnione w tekście. Jeśli ich znaczenie nie jest Ci znane, zapoznaj się proszę z ich szerszym omówieniem w ogólnodostępnych źródłach

Komunikacja skrypt-program

Skrypty pisane w Bashu mogą nie tylko wywoływać polecenia wbudowane Basha, ale też inne programy, w tym programy napisane przez nas samych. Istotnym elementem jest prawidłowa komunikacja skryptu z programem. Skupimy się przede wszystkim na przekazywaniu parametrów i odbieraniu wyników działania programu, w tym informacji o prawidłowym bądź nieprawidłowym zakończeniu jego działania.

System kontroli wersji

Proces tworzenia oprogramowania jest zwykle czasochłonny i przebiega w wieloosobowych zespołach. Wymaga to podejścia przyrostowego: zaczynamy od małego fragmentu kodu, sprawdzamy czy napisana część działa zgodnie z założeniami, dopisujemy kolejny fragment itd; oraz interakcji z innymi twórcami, którzy w tym samym czasie pracują nad tym samym projektem. Bez usystematyzowanego podejścia w takim procesie praca jest właściwie niemożliwa. Stąd, powstało wiele narzędzi wspomagających zarządzanie kolejnymi wersjami kodu, sprawdzanie poprawności wprowadzonych zmian i ułatwiających interakcję pomiędzy twórcami. Jednym z takich narzędzi jest *Git*, którego podstawowe komendy omówimy w tym kursie.

System składu tekstu

Oprócz kodu programu możemy potrzebować wytworzyć również jego dokumentację lub inne dokumenty tekstowe. Wtedy zamiast edytora tekstu typu WYSIWYG (ang. *what you see is what you get*), np. takiego jak LibreOffice Writer, możemy skorzystać z systemu składu tekstu opartego na języku znaczników. Dzięki takiemu podejściu możemy łatwiej oddzielić treść od stylu, utrzymać spójny wygląd całego dokumentu, w prosty sposób generować dokumenty programem oraz stosować system kontroli wersji do zarządzania dokumentami jeśli są tworzone manualnie. Jednym z takich systemów jest LaTeX – systemem makr i poleceń, który ułatwia skład publikacji elektronicznych przy pomocy programu TeX.

Inne narzędzia

To oczywiście nie koniec możliwych narzędzi w warsztacie programisty, jednakże tutaj skupimy się na tych kilku wybranych, budując stopniowo umiejętności wykorzystania każdego z nich.

Warsztat fachowca to nie jest coś, czego można się nauczyć raz na zawsze. Zwłaszcza w dziedzinie, która się rozwija, takiej jak informatyka, również narzędzia są cały czas udoskonalane. Powstają ich nowe rodzaje oraz sposoby wykorzystania. Tak więc każdy, kto chce być produktywny i tworzyć rozwiązania wysokiej jakości, powinien sukcesywnie rozbudowywać swój warsztat i umiejętności jego wykorzystania nie tylko w czasie studiów, ale w ciągu całej profesjonalnej kariery.

Zawartość podręcznika

Podręcznik składa się z dwóch głównych części. Pierwsza z nich traktuje o elementach systemu Linux niezbędnych do poruszania się w strukturze katalogów, zarządzania plikami i uruchamianymi programami. Część druga omawia w podstawowym zakresie narzędzia użyteczne do pracy z projektem programistycznym. Na koniec proponujemy kilka projektów o różnym stopniu złożoności, które mogą posłużyć do przećwiczenia przyswojonych umiejętności.

Rozdział 2

Linux i elementy Basha

W tym rozdziale poznamy elementy powłoki Bash, napiszemy pierwszy skrypt i uruchomimy go. Zapoznamy się z koncepcją strumieni danych, możliwością ich przekierowania i tworzenia potoków. Następnie zobaczymy, jak uruchamiać procesy w tle, monitorować ich wykonanie i nimi zarządzać. Na koniec zajmiemy się wpływaniem na działanie programów poprzez zmienne środowiskowe i parametry.

2.1 Elementy Basha

Zaczniemy od poruszania się po strukturze katalogów i podstawowych operacjach na plikach bez *interfejsu graficznego*. Posłuży nam do tego *terminal*, czyli program, który otwiera okno pozwalające na komunikację z powłoką Linuxa. Pobiera on polecenia z klawiatury i przekazuje je systemowi operacyjnemu do wykonania jako instrukcje.

Korzeniem drzewa katalogów (czyli katalogiem głównym) w systemie Linux jest `/` (ukośnik). W nim znajdują się wszystkie pozostałe katalogi systemu, w tym katalog domowy użytkownika – zazwyczaj noszący taką samą nazwę jak login, którym posługujemy się przy logowaniu do systemu. Każdy katalog ma swój adres w systemie plików, nazywany ścieżką. *Ścieżka bezwzględna* (ang. *absolute path*) to pełny adres katalogu, zaczynający się od katalogu głównego.

Przykładowo, dla użytkownika o loginie `jkowalski`, ścieżką bezwzględną do jego katalogu domowego będzie: `/home/jkowalski`. W terminalu Linuxa przed znakiem zachęty (promptem, oznaczanym zwykle jako $) widoczna jest ścieżka do katalogu, w którym aktualnie się znajdujemy. Dzięki temu możemy łatwo sprawdzić, z jakim katalogiem powiązane są wykonywane w danym momencie polecenia powłoki Bash. Po zalogowaniu do systemu terminal zwykle wyświetla linię w postaci: `login@nazwa_komputera:~$`. Znak ~ (tylda) oznacza katalog domowy użytkownika i jest skrótem dla pełnej ścieżki `/home/nazwa_użytkownika`. Dopóki nie zmienimy bieżącego katalogu, wszystkie wykonywane przez nas polecenia będą dotyczyć właśnie tego katalogu domowego.

2.1.1 Podstawowe komendy powłoki Bash

Operacje na plikach mogą być wykonywane przy pomocy poleceń takich jak:

- `pwd` – wypisanie pełnej ścieżki do bieżącego katalogu (ang. ***p**rint **w**orking **d**irectory*)
- `ls` – wypisanie informacji o plikach (ang. ***ls**t*)
- `cd` – zmiana katalogu (ang. ***c**hange **d**irectory*)
- `rm` – usunięcie plików (ang. ***rem**ove*)

- `mkdir` – utworzenie katalogu (ang. ***mk**e **dir**ectory*)
- `cp` – kopiowanie plików i katalogów (ang. ***cp**y*)
- `mv` – przenoszenie oraz zmiana nazwy plików i katalogów (ang. ***mv**e*)

Pomoc dla konkretnej komendy można uzyskać korzystając z poleceń `man` lub `info` pisząc na przykład: `man mkdir` albo podając opcję `--help`, np: `mkdir --help`.

Ćwiczenie 1. Utwórz pusty katalog, i wypróbuj polecenie `ls` z opcją `-a`. Np:

```
mkdir TempDir
ls -a
```

Opcja `-a` wypisuje również pliki, których nazwa zaczyna się od kropki (domyślnie są one pomijane). Widzimy też nazwy plików specjalnych `.` i `..`. Pierwszy z nich (pojedyncza kropka) oznacza katalog bieżący, a drugi (dwie kropki) – katalog nadrzędny.

Ćwiczenie 2. Wypróbuj polecenia: `cd .` i `cd ..` – pierwsze z nich zmienia katalog bieżący na bieżący (czyli właściwie nie ma skutku), a drugie na katalog nadrzędny.

Ćwiczenie 3. Stwórz katalog `Warsztat2025` w katalogu domowym $\sim$, a w nim dwa kolejne katalogi: `laboratoria` oraz `wyklady` oraz katalog `laboratorium1` w katalogu `laboratoria`. Następnie użyj polecenia `tree Warsztat2025`, aby zobaczyć znajdujące się tam drzewo podkatalogów.

Wydruk powinien przypominać poniższy:

```
Warsztat2025/
|--laboratoria
|   |--laboratorium1
|--wyklady

3 directories, 0 files
```

Ścieżka względna (ang. *relative path*) to ścieżka określająca położenie pliku względem innego katalogu. Korzystając z poleceń Basha, możemy podawać ścieżki względem katalogu bieżącego.

Ćwiczenie 4. Stwórz pusty plik tekstowy `plik1.txt` (polecenie `touch plik1.txt`) w katalogu `Warsztat2025`. Następnie przenieś stworzony plik do katalogu `laboratorium1`.

2.1.2 Prawa dostępu

Każdy plik i katalog w systemie Linux ma przypisane prawa dostępu. Dotyczą one trzech podmiotów: właściciela (ang. *user*), grupy (ang. *group*) i pozostałych użytkowników (ang. *others*).

Ćwiczenie 5. Wyświetl informacje o pliku `plik1.txt` za pomocą polecenia: `ls -l`. Pamiętaj o odpowiedniej ścieżce do pliku.

Opcja `-l` oznacza długi format wypisywania informacji o plikach. Przykładowy początek wydruku może wyglądać tak:

```
-rw-------+ 1 myname  ...
```

Pierwsze dziesięć znaków to atrybuty pliku oznaczające:

- typ pliku – pierwszy;
- prawa dostępu dla właściciela – drugi, trzeci, czwarty;
- prawa dostępu dla grupy – piąty, szósty, siódmy;
- prawa dostępu dla pozostałych użytkowników – ósmy, dziewiąty, dziesiąty.

W Linuxie **prawie wszystko jest traktowane jako plik** (o konkretnych właściwościach), także urządzenia czy terminale. Każdy plik to zbiór znaków, pogrupowanych w wiersze (oddzielone znakiem \n), zakończony znakiem końca pliku EOF. Na potrzeby zajęć będziemy rozróżniać zwykłe pliki (**"-"** w miejscu znaku typu pliku) oraz katalogi (**"d"**).

Ćwiczenie 6. Wypróbuj `ls -l` na dowolnym pliku, a potem katalogu. Zaobserwuj różnice w wydrukach.

Dla zwykłego pliku symbole `r,w,x` oznaczają kolejno:

- odczyt (ang. *read*) – możliwość wyświetlenia zawartości pliku;
- zapis (ang. *write*) – mozliwość zapisania zmian w pliku;
- wykonanie (ang. *execute*) – możliwość uruchomienia (programy binarne i skrypty).

Ćwiczenie 7. Stwórz plik `plik2.txt` w katalogu `laboratorium1`. Użyj polecenia: `chmod o+r`, aby dodać prawo do odczytu dla pozostałych użytkowników. Odpowiednim poleceniem sprawdź, czy rzeczywiście to prawo zostało nadane.

2.1.3 Pierwszy skrypt

Ćwiczenie 8. Wpisz polecenia z ćwiczenia 1 do pliku tekstowego `pierwszy.sh`. Możesz to zrobić przy pomocy dowolnego edytora, np.: `Nano` lub `VSCode`.

W ten sposób utworzyliśmy skrypt, czyli plik zawierający ciąg poleceń danej powłoki. Aby go uruchomić (wykonać) należy wpisać polecenie: `./pierwszy.sh`.
UWAGA: `./` oznacza, że chcemy uruchomić skrypt znajdujący się w bieżącym katalogu. Jeśli plik skryptu jest gdzieś indziej, to musimy podać pełną ścieżkę.

Ćwiczenie 9. Spróbuj uruchomić powyższy skrypt. Co można zaobserwować?

Skrypt nie wykonał się, ponieważ domyślnie nie ma on ustawionego prawa do wykonywania (Sprawdź jego atrybuty poleceniem: `ls -l`).

Ćwiczenie 10. Za pomocą odpowiedniego polecenia ustaw atrybut wykonywalny dla właściciela pliku `pierwszy.sh`, a następnie uruchom swój skrypt.

Ćwiczenie 11. Stwórz kolejny skrypt `drugi.sh` z kilkoma poznanymi wcześniej pleceniami Basha. Wypróbuj jego działanie.

Aby skrypt wykonał się w Bashu, nawet jeśli używana jest inna powłoka, na początku skryptu dodajemy linię:

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

Teraz nasz skrypt wygląda następująco:

```
#!/bin/bash

mkdir TempDir
ls -a
```

2.1.4 Zmienne środowiskowe

Zmienna środowiskowa (ang. *environment variable*) to nazwana wartość, zazwyczaj zawierająca ciąg znaków, przechowywana i zarządzana przez powłokę. Niektóre zmienne są ustawiane przy starcie powłoki i nie można ich zmienić, np. `USER` (nazwa użytkownika) i `SHELL` (używana powłoka).
Aby wyświetlić zawartość jakiejś zmiennej spróbuj w terminalu wpisać na przykład: `echo $SHELL`. Spodziewany efekt to:

```
/bin/bash
```

wskazujący na lokalizację uruchomionej powłoki. Kolejne zmienne środowiskowe, które mogą być przydatne, to np.:

- `PATH` – lista lokalizacji, w których powłoka poszukuje programów do uruchomienia
- `USER` – nazwa użytkownika
- `HOME` – katalog domowy użytkownika
- `EDITOR` – domyślny edytor tekstu
- `LD_LIBRARY_PATH` – lista lokalizacji w których system poszukuje bibliotek

Inna ważna zmienna mówi o wartości zwróconej przez ostatnio wykonaną komendę. Spróbuj uruchomić poniższy skrypt:

```
ls
echo $?
rm jakisnieistniejacyplik.cos
echo $?
```

Wartość zero jest interpretowana jako wykonanie bezbłędne, a każda wartość inna niż zero oznacza błąd. W zależności od polecenia różne wartości oznaczają inną kategorię błędu.

Dzięki zmiennej środowiskowej `PATH` możemy łatwo sprawić, że nasz skrypt będzie można uruchamiać, wpisując jedynie jego nazwę, bez podawania pełnej ścieżki. Wykonaj polecenie `PATH="$PATH:sciezkabezwzglednadoskryptu"` oraz uruchom skrypt przez podanie wyłącznie jego nazwy.
UWAGA: Po zamknięciu terminala zmiana zmiennej `PATH` jest tracona i przywracany jest jej pierwotny stan.

Ćwiczenie 12. Skompiluj i uruchom poniższy program w C. Po jego wykonaniu sprawdź wartość zmiennej `$?` – wartość ta powinna odpowiadać wartości podanej po instrukcji `return` w programie.

```
int main() {
    return 7;
}
```

Ćwiczenie 13. Zmień wartość zwracaną przez program z ćwiczenia 12 na `-1`, a następnie na `257`. Za każdym razem skompiluj program i sprawdź wartość zmiennej `$?`. Wartości, o których możemy być pewni, że zostaną dokładnie przekazane, ograniczają się do zakresu `[0, 255]`.

Ćwiczenie 14. Sprawdź wartość innych zmiennych środowiskowych w swoim systemie (polecenie `printenv` drukuje ich nazwy).

2.2 Standardowe strumienie danych

Strumienie standardowe (ang. *standard streams*) stanowią podstawowy mechanizm wymiany danych pomiędzy programem komputerowym a jego środowiskiem. Są to trzy nazwane kanały:

- standardowe wejście (`stdin`),
- standardowe wyjście (`stdout`) oraz
- standardowy strumień błędów (`stderr`).

Gdy program wykonywany jest za pośrednictwem powłoki (tak jak w Bashu), strumienie są połączone z terminalem tekstowym, w którym działa powłoka, i z klawiaturą. Ustawienia te można zmienić za pomocą przekierowania lub potoku (ang. *pipe*). Typowo proces potomny dziedziczy standardowe strumienie swojego procesu nadrzędnego.

Strumienie standardowe mają też przypisane numery: `0`,`1` i `2` odpowiednio dla `stdin`; `stdout`; `stderr`. Są to tak zwane *deskryptory* – identyfikatory pliku wykorzystywane przez system operacyjny. Po wykonaniu operacji otwarcia, deskryptor może być wykorzystywany wielokrotnie przez wywołania systemowe w operacjach wejścia/wyjścia. Po uruchomieniu procesu deskryptory pliku standardowej komunikacji są już otwarte i można na nich operować.

2.2.1 Przekierowanie strumieni

Przekierowanie standardowego wyjścia odbywa się za pomocą operatorów `>` lub `>>`. Polecenie `ls > pliki.txt` wydane w konsoli spowoduje przekierowanie wyniku działania polecenia `ls` do pliku `pliki.txt`, zamiast wyświetlenia go w oknie terminala.

Warto zwrócić uwagę, że polecenie wykona się prawidłowo, nawet jeśli wcześniej plik `pliki.txt` nie istniał. Dzieje się tak dlatego, że plik ten zostaje utworzony przed wykonaniem polecenia `ls`. Jeżeli natomiast plik istniał, to jego zawartość zostanie zastąpiona. Jeśli chcemy, aby dane zostały dopisane na koniec pliku, używamy operatora `>>`, np.: `ls >> pliki.txt`.

Ćwiczenie 15. Wypróbuj sekwencję poniższych komend (spróbuj przewidzieć wynik działania każdej z nich przed jej uruchomieniem):

```
echo "to" > tekst.txt
echo napis > tekst.txt
echo "to␣napis" >> tekst.txt
```

Podobnie możemy przekierować standardowy strumień wejściowy, tym razem używając operatora `<` (znak mniejszości). Jeśli w pliku `files.txt` znajduje się wynik działania komendy `ls`, możemy teraz policzyć wypisane pliki przy pomocy komendy `wc` (ang. ***w**ord **c**ount*) pisząc:

```
wc < pliki.txt
```

Program `wc` wczyta dane z pliku zamiast ze standardowego wejścia.

Ćwiczenie 16. Napisz program w C, który wykrywa podwójne spacje (dwa kolejne znaki) wczytywane ze standardowego wejścia. Uruchom swój program przekierowując strumień wejściowy z pliku tekstowego.

2.2.2 Tworzenie potoków

Potok (ang. *pipe*) łączy `stdout` jednego procesu z `stdin` drugiego procesu, a więc przekierowuje strumień wyjściowy jednego procesu, podając go jako strumień wejściowy kolejnego procesu. Pisząc więc na przykład:

```
ls | wc
```

przekierowujemy strumień wyjściowy polecenia `ls` jako strumień wejściowy dla `wc`.

Możemy także łączyć ze sobą wiele poleceń (procesów), pisząc znak `|` po każdej z komend. Na przykład:

```
ls | grep hej | wc
```

policzy pliki, które mają podsłowo `hej` w nazwie przekierowując wyjście `ls` jako strumień wejściowy dla programu `grep`, który przepisze na wyjście tylko linie zawierające `hej`. Właśnie dokładnie te linie trafią jako wejście do `wc`.

2.3 Programy i procesy

Przez *program* rozumiemy ciąg poleceń dla maszyny (komputera). Rozróżniamy *program wykonywalny* – ciąg poleceń w postaci skompilowanej, trudnej do przeczytania i zrozumienia dla człowieka i *kod źródłowy* – program napisany w *języku programowania*, który możemy wykonać po kompilacji lub przy pomocy *interpretera*, który jest w stanie wykonać komendy zapisane w programie.

Przez *proces* rozumiemy instancję działającego programu, która powstała w wyniku jego uruchomienia. W danej chwili może istnieć i działać wiele różnych procesów, będących wynikiem uruchomienia tego samego programu. Każdy proces posiada numer przydzielony przez system operacyjny –PID (ang. ***p**rocess **i**dentification **n**umber*), i jest wykonywany niezależnie od innych procesów działających na tym samym komputerze. Możliwa jest jednak komunikacja pomiędzy procesami za pomocą mechanizmów dostarczanych przez system operacyjny i stąd *aplikacje* (programy użytkowe) mogą składać się z wielu programów i procesów współpracujących ze sobą.

2.3.1 Wywołania w tle

Dodanie do komendy znaku `&` powoduje, że komenda jest wykonana w *w tle* (ang. *background*), to znaczy nie blokuje konsoli w trybie interaktywnym, a w skrypcie wykonywane są kolejne polecenia bez oczekiwania na zakończenie działania komendy.

Ćwiczenie 17. Wypróbuj: `sleep 5` i `sleep 5 &`.

2.3.2 Informacje o procesach

Polecenie `ps` wyświetla informacje o wybranych aktywnych procesach. Wypróbuj komendę `ps --help` żeby zobaczyć skróconą wersję opisu programu oraz `man ps` i `info ps`.

Ćwiczenie 18. Porównaj poniższe dwa programy i spróbuj je uruchomić:

```
#include<unistd.h>
#include<stdio.h>

int main() {
  printf("Hej\n");
  sleep(10);
  printf("Ho\n");
  return 0;
}
```

```
#!/bin/bash

echo Hej
sleep 10
echo Ho
```

Oba robią mniej więcej to samo: po wypisaniu pierwszego komunikatu program jest wstrzymywany, a następnie wypisywany jest drugi komunikat. Przypuśćmy, że pierwszy program po kompilacji nazywa się `hejho`, a drugi skrypt to `hejho.sh`. Po uruchomieniu obu programów w tle i wywołaniu `ps`:

```
./hejho &
./hejho.sh &
ps
```

można się spodziewać informacji podobnej do poniższej:

```
    PID TTY          TIME CMD
 555115 pts/4    00:00:00 bash
 605697 pts/4    00:00:00 hejho
 605771 pts/4    00:00:00 hejho.sh
 605772 pts/4    00:00:00 sleep
 605797 pts/4    00:00:00 ps
```

Widzimy, że powyższe wywołania stworzyły dodatkowy proces `sleep`.

Ćwiczenie 19. Sprawdź, który z powyższych programów spowodował uruchomienie osobnego procesu `sleep`.

Ćwiczenie 20. Sprawdź pomoc dla komendy `sleep`, to znaczy wypróbuj następujące polecenia:

```
man sleep
apropos sleep
info sleep
man -S3 sleep
```

Zwróć uwagę na różnicę pomiędzy komendą `sleep` wywołaną w Bashu i funkcją `sleep` zadeklarowaną w pliku nagłówkowym `unistd.h`.

Ćwiczenie 21. Wypróbuj komendy `ps` i `ps | wc`.

Jeśli masz wątpliwości, jakie liczby wypisze `wc` w drugim przypadku, to masz rację. Oba programy są uruchamiane równolegle i nie mamy gwarancji, czy `ps` zdąży zarejestrować działanie `wc`, czy też nie.

2.3.3 Unicestwienie działającego procesu

Jeśli działający program nie zachowuje się tak, jak planowaliśmy, to możemy go unicestwić komendą `kill` podając numer procesu, np.: `kill -9 605697` wyśle sygnał o numerze 9 (SIGKILL) do procesu o numerze 605697 powodując jego unicestwienie. Listę innych zdefiniowanych sygnałów możemy zobaczyć przy pomocy polecenia `kill -l`.

2.4 Przekazywanie parametrów

Znamy już standardowe strumienie danych i wiemy, jak je przekierowywać. Teraz skupimy się na sposobie dostarczania danych do skryptów w Bashu oraz programów napisanych w C za pomocą parametrów.

2.4.1 Zmienne

Proces interpretera poleceń (podobnie, jak inne procesy) przechowuje w swojej pamięci pewną liczbę zmiennych wraz z przyporządkowanymi im wartościami. Zarówno nazwy zmiennych, jak i przyporządkowane im wartości, są łańcuchami znaków (napisami). W pamięci procesu są one przechowywane w postaci listy par:

```
zmienna_1=wartość_1
zmienna_2=wartość_2
.....................
zmienna_n=wartość_n
```

Użytkownik interpretera poleceń ma możliwość wyświetlania listy zmiennych oraz jej modyfikacji (dopisywania nowych zmiennych, usuwania zmiennych oraz zmiany ich wartości). Nową zmienną tworzymy za pomocą przypisania `nazwa_zmiennej=wartość`, podobnie zmieniamy jej wartość, a odwołujemy się do niej używając znaku dolara: `$nazwa_zmiennej`. Zbiór wszystkich zmiennych możemy wyświetlić za pomocą polecenia `set`, a usuwamy jakąś zmienną za pomocą komendy `unset nazwa_zmiennej`.

Ćwiczenie 22. Zadeklaruj zmienną o dowolnej nazwie z dowolną wartością i wyświetl ją w konsoli za pomocą polecenia: `echo $nazwa_zmiennej`. Następnie zmień jej wartość i usuń odpowiednim poleceniem. Upewnij się, że nie ma jej już wśród zmiennych, np. poleceniem `set | grep $nazwa_zmiennej`.

Bash umożliwia operacje na łańcuchach znaków stanowiących wartości zmiennych: obliczanie długości, złączanie (konkatenacja), wycinanie podłańcuchów, sprawdzanie zgodności z podanym wzorcem i inne.

Ćwiczenie 23. Utwórz dwie zmienne i wykonaj polecenie: `zmienna=$zmienna_1$zmienna_2`, a następnie polecenia: `echo $zmienna` oraz `echo ${zmienna:2}`.

Jeśli wartości zmiennych są poprawnymi zapisami liczb całkowitych (w sensie reguł składniowych Basha), to jest możliwe wykonywanie na nich operacji arytmetycznych (z typowego zbioru udostępnianego przez języki programowania, w zakresie arytmetyki liczb całkowitych).

Ćwiczenie 24. Zadeklaruj kilka zmiennych numerycznych (liczbowych) pisząc: `declare -i zmienna_1 zmienna_2 ... zmienna_n`

Następnie przyporządkuj im wartości całkowitoliczbowe pisząc `zmienna=liczba` i wypróbuj możliwość wykonywania obliczeń arytmetycznych przy użyciu zadeklarowanych zmiennych.

2.4.2 Zmienne środowiskowe

Niektóre spośród wszystkich zmiennych danego procesu (uruchomionej powłoki Basha) są oznakowane jako zmienne środowiska. Wcześniej omówiliśmy już np. działanie takich zmiennych jak $PWD czy $USER. Zmienne środowiska są dziedziczone przez wszystkie procesy potomne danego procesu (w momencie uruchomienia procesu potomnego są kopiowane do jego środowiska). Użytkownik Basha ma możliwość zaznaczania zmiennych jako należących do środowiska (czyli przeznaczonych do dziedziczenia), jak również usuwania tego oznakowania.

Ćwiczenie 25. Sprawdź, że polecenie `set` wyświetla wcześniej utworzone zmienne, a polecenie `printenv` – nie wyświetla. Następnie użyj polecenia `export zmienna` do umieszczenia jakiejś nowo utworzonej zmiennej w środowisku (wyeksportowania). Sprawdź, że polecenie `printenv` będzie ją teraz wyświetlało.

Ćwiczenie 26. Wypróbuj działanie polecenia `export -n zmienna` (usunięcie ze środowiska). Uruchom nową kopię Basha (polecenie bash) i sprawdź, że odziedziczone zostały tylko te zmienne, które aktualnie były w środowisku pierwotnego basha. Na koniec wyłącz kopię basha (polecenie `exit`).

2.4.3 Przekazywanie wartości zmiennych

Wartości zmiennych w Bashu możemy przekazywać do skryptu bądź programu. Każdy skrypt oraz program w C może pobierać dane, które nazywamy parametrami lub argumentami. Dla skryptu standardowo podajemy kolejne parametry po jego nazwie: `./nazwa_skryptu parametr_1 parametr_2 ... parametr_n`. W kodzie skryptu odwołujemy się do argumentów przez zmienną $\$n$, gdzie n jest liczbą porządkową argumentu (pierwszy argument to `$1`, `$0` zawiera nazwę uruchomionego skryptu, a dokładnie ścieżkę). Ponadto mamy także dodatkowe zmienne:

- `$#` – liczba wszystkich argumentów
- `$*` – wszystkie argumenty

Ćwiczenie 27. Uruchom poniższy skrypt kilka razy z różnymi wejściowymi parametrami:

```
#!/bin/bash

echo "Nazwa␣programu␣to:␣$0"
echo "Liczba␣argumentów:␣$#"
echo "Podane␣argumenty␣to:␣$*"
```

W języku C argumenty (parametry) są przekazywane bezpośrednio do głównej funkcji `main()` pobierającej 2 argumenty, które zwyczajowo noszą nazwy `argc` i `argv`. Pierwszy przechowuje liczbę całkowitą reprezentującą liczbę przekazanych argumentów, a drugi jest tablicą znaków. Deklaracja funkcji wygląda więc następująco:

```
int main(int argc, char **argv)
```

Ćwiczenie 28. Uruchom poniższy program w C z dwoma argumentami i zauważ różnicę w liczbie argumentów w stosunku do skryptu:

```
#include<stdio.h>

int main(int argc, char **argv){
        printf("Nazwa␣programu␣to:␣%s\n", argv[0]);
        printf("Liczba␣argumentów:␣%d\n", argc);
        printf("Podane␣2␣argumenty␣to␣:␣%s␣%s\n", argv[1], argv[2]);
        return 0;
}
```

Rozdział 3

Tworzenie i prowadzenie projektów programistycznych

W tym rozdziale omówimy proces budowania projektu i ideę jego automatyzacji. Przyjrzymy się wybranym opcjom kompilatora, poznamy elementy systemów `make` i `cmake`. Zapoznamy się z ideą systemu kontroli wersji i nauczymy się pracować z systemem `git` w najprostszym scenariuszu, gdy nad projektem pracuje jedna osoba. Następnie stworzymy bibliotekę i poznamy różnicę pomiędzy bibliotekami statycznymi i dynamicznymi. Stworzymy bibliotekę w ramach własnego projektu oraz użyjemy biblioteki zewnętrznej. Na koniec zapoznamy się z wybranymi metodami usuwania błędów i narzędziami, które to ułatwiają.