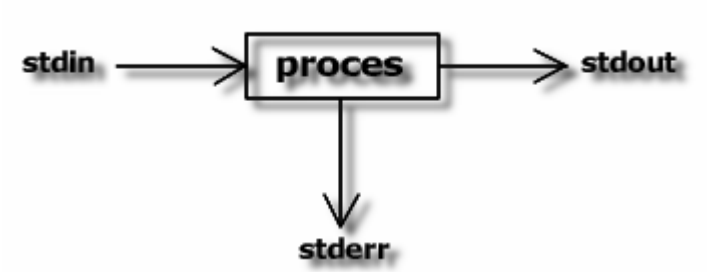
Strumienie danych

Strumienie danych

Każdy uruchomiony w Linuxie proces skądś pobiera dane, gdzieś wysyła wyniki swojego działania i komunikaty o błędach. Tak więc procesowi przypisane są trzy strumienie danych:

- **stdin** (ang. *standard input*) czyli **standardowe wejście**, skąd proces pobiera dane, domyślnie jest to **klawiatura**
- **stdout** (ang. *standard otuput*) to **standardowe wyjście**, gdzie wysyłany jest wynik działania procesu, domyślnie to **ekran**
- **stderr** (ang. *standard error*) **standardowe wyjście błędów**, tam trafiają wszystkie komunikaty o błędach, domyślnie **ekran**

Rys.Strumienie danych



Linux wszystko traktuje jako plik, niezależnie od tego czy to jest plik zwykły, katalog, urządzenie blokowe (klawiatura, ekran) itd. Nie inaczej jest ze strumieniami, **BASH** identyfikuje je za pomocą przyporządkowanych im liczb całkowitych (od **0** do **2**) tak zwanych **deskryptorów plików**.
I tak:

- **0** to plik z którego proces pobiera dane **stdin**
- **1** to plik do którego proces pisze wyniki swojego działania **stdout**
- **2** to plik do którego trafiają komunikaty o błędach **stderr**

Za pomocą **operatorów przypisania** można manipulować strumieniami, poprzez przypisanie deskryptorów: **0**, **1**, **2**innym plikom, niż tym reprezentującym klawiaturę i ekran.

- **Przełączanie standardowego wejścia**

  Zamiast klawiatury jako standardowe wejście można otworzyć plik:

  ```
  < plik
  ```

  Przykład:

  Najpierw stwórzmy plik **lista** o następującej zawartości:

  ```
  slackaware
  redhat
  debian
  caldera
  ```

  Użyjemy polecenia **sort** dla którego standardowym wejściem będzie nasz plik.

  ```
  sort < lista
  ```

  Wynikiem będzie wyświetlenie na ekranie posortowanej zawartość pliku **lista**:

```
caldera
debian
redhat
slackware
```

- **Przełączanie standardowego wyjścia**

  Wynik jakiegoś polecenia można wysłać do pliku, a nie na ekran, do tego celu używa się operatora:

  ```
  > plik
  ```

  Przykład:

  ```
  ls -la /usr/bin > ~/wynik
  ```

  Rezultat działania polecenia **ls -la /usr/bin** trafi do pliku o nazwie **wynik**, jeśli wcześniej nie istniał plik o takiej samej nazwie, to zostanie utworzony, jeśli istniał cała jego poprzednia zawartość zostanie nadpisana.

  Jeśli chcemy aby dane wyjściowe dopisywane były na końcu pliku, bez wymazywania jego wcześniejszej zawartości, stosujemy operator:

  ```
  >> plik
  ```

  Przykład:

  ```
  free -m >> ~/wynik
  ```

  Wynik polecenia **free -m** (pokazuje wykorzystanie pamięci RAM i swap'a) zostanie dopisany na końcu pliku**wynik**, nie naruszając jego wcześniejszej zawartości.

- **Przełączanie standardowego wyjścia błędów**

  Do pliku można też kierować strumień diagnostyczny:

  ```
  2> plik
  ```

  Przykład:

  ```
  #!/bin/bash

  echo "Stderr jest skierowane do pliku error"
  ls -y 2> ~/error    #błąd
  ```

  W powyższym skrypcie polecenie **ls** jest użyte z błędną opcją **-y**, komunikat o błędzie trafi do pliku **error**.

  Za pomocą operatora:

  ```
  2>> plik
  ```

  można dopisać do tego samego pliku kilka komunikatów o błędach, dopisanie kolejnego nie spowoduje skasowania wcześniejszej zawartości pliku.

Przykład:

```
#!/bin/bash

echo "Stderr jest skierowane do pliku error"
ls -y 2> ~/error              #błąd
cat /etc/shadow 2>> ~/error   #błąd2
```

Jako błąd drugi zostanie potraktowane polecenie **cat /etc/shadow** (zakładając, że zalogowałeś się jako **użytkownik**) ponieważ prawo **odczytu** pliku **/etc/shadow** ma tylko **root**.

Polecenie read

Polecenie read

Czyta ze standardowego wejścia pojedynczy wiersz.

Składnia:

```
read -opcje nazwa_zmiennej
```

Przykład:

```
#!/bin/bash
echo -n -e "Wpisz coś:\a"

read wpis
echo "$wpis"
```

To co zostało wpisane trafi do zmiennej **wpis**, której to wartość czyta polecenie **read wpis**, zmienna nie musi być wcześniej tworzona, jeśli istniała wcześniej, jej zawartość zostanie zastąpiona tym co wpisaliśmy.

Przykład:

```
#!/bin/bash
echo "Wpisz coś:"

answer="napis"
read answer
echo "$answer"
```

Wcześniejsza wartość zmiennej **answer** została zastąpiona.

Polecenie **read** pozwala na przypisanie kilku wartości kilku zmiennym.

Przykład:

```
#!/bin/bash
echo "Wpisz cztery wartości:"

read a b c
echo "Wartość zmiennej a to: $a"
echo "Wartość zmiennej b to: $b"
echo "Wartość zmiennej c to: $c"
```

Nie przypadkiem w powyższym przykładzie pojawiło się polecenie wpisania czterech wartości, pierwsza wartość trafi do zmiennej **a**, druga do zmiennej **b**, natomiast trzecia i czwarta oraz rozdzielające je znaki separacji przypisane zostaną zmiennej **c**.

Wybrane opcje:

- **-p**

  Pokaże znak zachęty bez kończącego znaku nowej linii.

```
#!/bin/bash

read -p "Pisz:" odp
echo "$odp"
```

- **-a**

  Kolejne wartości przypisywane są do kolejnych indeksów zmiennej tablicowej.

  Przykład:

```
#!/bin/bash
echo "Podaj elementy zmiennej tablicowej:"

read tablica
echo "${tablica[*]}"
```

- **-e**

  Jeśli nie podano żadnej nazwy zmiennej, wiersz trafia do **$REPLY**.

  Przykład:

```
#!/bin/bash
echo "Wpisz coś:"

read -e
echo "$REPLY"
```

Instrukcja if

Instrukcja warunkowa if
Sprawdza czy warunek jest prawdziwy, jeśli tak to wykonane zostanie polecenie lub polecenia znajdujące się po słowie kluczowym **then**. Instrukcja kończy się słowem **fi**.

Składnia:

```
if warunek
then
  polecenie
fi
```

Przykład:

```
#!/bin/bash
if [ -e ~/.bashrc ]
then
  echo "Masz plik .bashrc"
fi
```

Najpierw sprawdzany jest warunek: czy istnieje w twoim katalogu domowym plik **.bashrc**, zapis **~/** oznacza to samo co **/home/twój_login** lub **$HOME**. Jeśli sprawdzany warunek jest prawdziwy to wyświetlony zostanie napis **Masz plik .bashrc**. W przeciwnym wypadku nic się nie stanie.

W sytuacji gdy test warunku zakończy się wynikiem negatywnym można wykonać inny zestaw poleceń, które umieszczamy po słowie kluczowym **else**:

Składnia:

```
if warunek
then
  polecenie1
else
  polecenie2
fi
```

Przykład:

```
#!/bin/bash
if [ -e ~/.bashrc ]
then
  echo "Masz plik.bashrc"
else
  echo "Nie masz pliku .bashrc"
fi
```

Jeśli warunek jest fałszywy skrypt poinformuje Cię o tym.
Można też testować dowolną ilość warunków, jeśli pierwszy warunek nie będzie prawdziwy, sprawdzony zostanie następny, kolejne testy warunków umieszczamy po słowie kluczowym **elif**.

Składnia:

```
if warunek
then
  polecenie1
elif warunek
then
  polecenie2
fi
```

Przykład:

```
#!/bin/bash
if [ -x /opt/kde/bin/startkde ]; then
   echo "Masz KDE w katalogu /opt"
elif [ -x /usr/bin/startkde ]; then
  echo "Masz KDE w katalogu /usr"
elif [ -x /usr/local/bin/startkde ]; then
  echo "Masz KDE w katalogu /usr/local"
else
  echo "Nie wiem gdzie masz KDE"
fi
```

Ten skrypt sprawdza gdzie masz zainstalowane **KDE**, sprawdzane są trzy warunki, najpierw czy plik wykonywalny **startkde** znajduje się w katalogu **/opt/kde/bin** jeśli go tam nie ma, szukany jest w **/usr/bin**, gdy i tu nie występuje sprawdzany jest katalog **/usr/local/bin**.

test

Jak się sprawdza warunki?

Służy do tego polecenie test. (**Uwaga!** Nie można skryptom nadawać nazwy **test**! Nie będą działać.)

Składnia:

```
test wyrażenie1 operator wyrażenie2
```

lub może być zapisane w postaci nawiasów kwadratowych:

```
[ wyrażenie1 operator wyrażenie2 ]
```

Uwaga! Między nawiasami a treścią warunku muszą być spacje, tak jak powyżej.

Polecenie **test** zwraca wartość 0 (true) jeśli warunek jest spełniony i wartość 1 (false) jeśli warunek nie jest spełniony. A gdzie jest umieszczana ta wartość? W zmiennej specjalnej **$?**.

A to kilka przykładów operatorów polecenia **test**:

- **-a** plik istnieje
- **-b** plik istnieje i jest blokowym plikiem specjalnym
- **-** plik istnieje i jest plikiem znakowym
- **-e** plik istnieje
- **-h** plik istnieje i jest linkiem symbolicznym
- **=** sprawdza czy wyrażenia są równe
- **!=** sprawdza czy wyrażenia są różne
- **-n** wyrażenie ma długość większą niż 0
- **-d** wyrażenie istnieje i jest katalogiem
- **-z** wyrażenie ma zerową długość
- **-r** można czytać plik
- **-w** można zapisywać do pliku
- **-x** można plik wykonać
- **-f** plik istnieje i jest plikiem zwykłym
- **-p** plik jest łączem nazwanym
- **-N** plik istnieje i był zmieniany od czasu jego ostatniego odczytu
- **plik1 -nt plik2** plik1 jest nowszy od pliku2
- **plik1 -ot plik2** plik1 jest starszy od pliku2
- **-lt** mniejsze niż
- **-gt** większe niż
- **-ge** większe lub równe
- **-le** mniejsze lub równe

Więcej przykładów operatorów w: **man bash**.

Pętla while

Pętla while
Najpierw sprawdza warunek czy jest prawdziwy, jeśli tak to wykonane zostanie polecenie lub lista poleceń zawartych wewnątrz pętli, gdy warunek stanie się fałszywy pętla zostanie zakończona.

Składnia:

```
while warunek
do
polecenie
done
```

Przykład:

```
#!/bin/bash
x=1;
while [ $x -le 10 ]; do
echo "Napis pojawił się po raz: $x"
x=$[x + 1]
done
```

Sprawdzany jest warunek czy zmienna **x** o wartości początkowej 1 jest mniejsza lub równa 10, warunek jest prawdziwy w związku z czym wykonywane są polecenia zawarte wewnątrz pętli: **echo "Napis pojawił się po raz: $x"** oraz **x=$[x + 1]**, które zwiększa wartość zmiennej **x** o 1. Gdy wartość **x** przekroczy 10, wykonanie pętli zostanie przerwane.

Pętla until

Pętla until

Sprawdza czy warunek jest prawdziwy, gdy jest fałszywy wykonywane jest polecenie lub lista poleceń zawartych wewnątrz pętli, miedzy słowami kluczowymi **do** a **done**. Pętla **until** kończy swoje działanie w momencie gdy warunek stanie się prawdziwy.

Składnia:

```
until warunek
do
polecenie
done
```

Przykład:

```
#!/bin/bash
x=1;
until [ $x -ge 10 ]; do
echo "Napis pojawił się po raz: $x"
x=$[x + 1]
done
```

Mamy zmienną **x**, która przyjmuje wartość 1, następnie sprawdzany jest warunek czy wartość zmiennej **x** jest większa lub równa 10, jeśli nie to wykonywane są polecenia zawarte wewnątrz pętli. W momencie gdy zmienna **x** osiągnie wartość, 10 pętla zostanie zakończona.

Zbiór potrzebnych poleceń

shift

Skrypty możemy uruchamiać podając im argumenty. Następujące zmienne o specjalnych nazwach pozwalają odczytywać argumenty:

$# zwraca liczbę argumentów,
$0 zwraca nazwę pliku bieżącego programu,
$1, $2, ... zwraca odpowiednio pierwszy argument, drugi argument, itd.,
$@ rozwija się do listy wszystkich argumentów; przydatne jeśli chcemy przekazać wszystkie argumenty innemu programowi.

Jeśli chcemy mieć pewność, że każdy argument będzie osobnym słowem, należy użyć cudzysłowów: "$@"; ma to znaczenie na przykład wtedy, gdy istnieje argument, który zawiera spację.

Aby operować na dalszych argumentach pomocne jest polecenie shift, które usuwa pierwszy argument, a pozostałe przesuwa o jeden w lewo. Aby n-krotnie wywołać polecenie shift wystarczy podać mu to n jako argument: shift n.

Na przykład dla skryptu *test_arg.sh* o zawartości

```
#!/bin/sh
# Testowanie argumentów
echo "Uruchomiłeś program `basename $0`"
echo "Wszystkie: $@"
echo "Pierwsze trzy: '$1', '$2', '$3'"
shift 2
echo "shift 2"
echo "Wszystkie: $@"
echo "Pierwsze trzy: '$1', '$2', '$3'"
```

mamy efekt

```
bashtest@host:~$ ./test_arg.sh Raz Dwa    "To jest  zdanie" Cztery
Uruchomiłeś program test_arg.sh
Wszystkie: Raz Dwa To jest zdanie Cztery
Pierwsze trzy: 'Raz', 'Dwa', 'To jest  zdanie'
shift 2
Wszystkie: To jest  zdanie Cztery
Pierwsze trzy: 'To jest  zdanie', 'Cztery', ''
bashtest@host:~$
```

cat

wyświetla zawartość plików

Postać: `cat [opcje] [plik...]`

Przykład:

```
> cat /etc/passwd
```

Polecenie `cat` może też posłużyć do tworzenia plików tekstowych

```
> cat > pliktekstowy
to jest tekst
który zostanie umieszczony
w pliku o nazwie pliktekstowy
Aby zakończyć wciśnij Ctrl+ D
```

lub do łączenia kilku plików w jedną całość - rezultat można przekierować do pliku:

```
> cat pliktekstowy dane.txt > nowy.txt
```

more

wyświetla zawartość pliku strona po stronie

Postać: `more [opcje] plik`

Przykład:

```
> more /etc/passwd
```

wyświetli zawartość pliku `passwd`

```
> ls /bin | more
```

pozwala przejrzeć listę plików w katalogu `/bin`

less

wyświetl zawartość pliku strona po stronie

Postać: `less [opcje] plik`

Jest to ulepszona wersja polecenia `more` pozwalająca poruszać się po pliku zarówno w przód jak i w tył.

Przykład:

```
> less /etc/passwd
```

Programy `more` i `less` posiadają wiele funkcji dostępnych za pomocą skrótów klawiszowych o których możemy się dowiedzieć wciskając `h`. Inne przydatne funkcje uzyskamy wciskając: `q` - wyjście z programu, `/` - poszukuje *wyrażenia* w pliku.

head

wyświetla początek pliku

Postać: `head [opcje] plik...`

Przykład:

```
> head /etc/passwd
```

wyświetli 10 pierwszych linii w pliku `passwd`

Najważniejsze opcje:
`-n liczba` wyświetli określoną liczbę początkowych linii
`-c liczba` wyświetli określona liczbę początkowych znaków
Przykład:

```
> head -c 10 /etc/passwd
```

wyświetli 10 pierwszych znaków pliku `passwd`

```
> ls | head -n 3
```

wyświetli nazwy trzech pierwszych plików z bieżącego katalogu

tail

wyświetla koniec pliku

Postać: `tail [opcje] plik...`

Działanie i opcje takie same jak w poleceniu `head` z tą różnicą, że wyświetlane jest zakończenie pliku
Przykład:

```
> tail -n 4 /etc/passwd
```

wyświetli cztery ostatnie linie pliku `passwd`

cmp

porównuje pliki znak po znaku

Postać: `cmp [opcje] plik1 plik2`

Polecenie wyświetla pozycje pierwszej napotkanej różnicy w zawartości plików.
Przykład:

```
> cmp plik1.txt plik2.txt
plik1.txt plik2.txt różnią się: bajt 30 linia 2
```

Najważniejsze opcje: `-c` wypisuje różniące się znaki

diff

znajduje różnice pomiędzy plikami
Postać: `diff [opcje] plik1 plik2`

Przykład:

```
> diff plik1.txt plik2.txt
```

Wynikiem działania jest wyświetlenie fragmentów które są różne w obu plikach wraz z informacją jak należy zmienić pierwszy z plików aby otrzymać drugi
(`c` zamień, `d` usuń, `a` dodaj fragment tekstu).
Np.:
`1,10c2,5` oznacza, że należy zamienić linie od 1 do 10 w pierwszym pliku na tekst który występuje w liniach od 2 do 5 w drugim pliku.
`3a5` oznacza, że w linii trzeciej pierwszego pliku należy dodać 5 linię z drugiego pliku

wc

liczy ilość znaków, słów i linii w pliku

Postać: `wc [opcje] plik...`

Najważniejsze opcje:
`-c` drukuje liczbę znaków/bajtów w pliku
`-w` drukuje liczbę wyrazów w pliku
`-l` drukuje ilość linii w pliku
Przykład:

```
> wc -c dane.txt
```

wyświetli ilość bajtów zajętych przez plik
Przykład:

```
> wc -l *.txt
```

wyświetli liczbę linii we wszystkich plikach o rozszerzeniu .txt znajdujących się w bieżącym katalogu.

sort

sortuje zawartość pliku tekstowego

Postać: `sort [opcje] plik...`

Przykład:

```
> sort dane.txt > posortowane.txt
```

spowoduje posortowanie linii zawartych w pliku `dane.txt` i przesłanie wyniku do pliku `posortowane.txt`
Niektóre opcje:
`-r` odwraca kolejność sortowania
`-u` usuwa duplikaty
`-f` wyłącza rozróżnianie małych i dużych liter
`-n` sortowanie liczb (standardowo dane sortowane traktowane są jako ciągi znaków)
Przykład:

```
> du . | sort -n
```

wyświetli listę plików w bieżącym katalogu posortowaną według rozmiaru
`+liczba` pozwala pominąć przy sortowaniu określoną liczbę pól (pola standardowo są rozdzielone białymi znakami (przestarzała wersja))
`-k poz1[,poz2]` pozwala specyfikować względem którego pola (kolumny) chcemy sortować
`-t separator` używa podanego znaku jako separatora pól (kolumn)
Przykład:

```
> ls -l | sort +4 -n
```

wyświetli posortowaną listę plików według piątej kolumny otrzymanej za pomocą polecenia `ls -l`

```
> sort -k 5 -t : /etc/passwd
```

Wyświetli posortowana listę użytkowników (piąta kolumna pliku `passwd`, gdzie kolumny sa oddzielone dwukropkami).

cut

Wypisuje wybrane fragmenty linii

Postać: `cut [opcja]... [plik]...`

Niektóre opcje:
`-b N` wypisuje tylko podane bajty
`-f N` wypisuje tylko podane kolumny (standardowo separatorami kolumn są białe znaki)
`-d znak` użyj podanego znaku jako separatora kolumn
Przykład:

```
> cut -c 1 /etc/passwd
```

wyświetli tylko pierwszy znak z każdej linii.

```
> cut -c 4-7 plik
```

wyświetli znaki od 4-go do 7-go.

```
> cut -f 2- plik
```

Wyświetli linie bez pierwszej kolumny

```
> cut -d : -f 5 /etc/passwd
```

wyświetli imiona i nazwiska użytkowników (5 kolumna pliku gdzie kolumny oddzielone są dwukropkiem).

tr

Zamienia znaki wczytane ze standardowego wejścia.

Postać: `tr łańcuch1 łańcuch2`

```
tr -d łańcuch
tr -s łańcuch
```

Najważniejsze opcje:
`-d` usuń podane w łańcuchu znaki
`-s` usuń wielokrotne wystąpienia tych samych znaków
Przykład:

```
> echo $PATH | tr : ' '
```

wyświetla wartość zmiennej $PATH zastępując dwukropki spacjami.

```
> echo Witaj swiecie | tr ai ia
```

w podanym haśle zamienia literę 'i' na 'a' oraz literę 'a' na 'i'

```
> echo Witaj swiecie | tr [a-z] [A-Z]
```

zamienia małe litery na duże

```
> cat plik | tr -d ' '
```

usuwa spacje z pliku

```
> cat plik | tr -s ' '
```

usuwa powtórzenia spacji w pliku

sed

Edytor strumieniowy

Postać: `sed [-n] [-e skrypt] [opcja]... [plik]...`

Odczytuje kolejne linie ze strumienia wejściowego (lub pliku), dokonuje edycji zgodnie z podanym skryptem i wynik wyświetla na standardowym wyjściu.
Najważniejsze opcje:
`-n` hamuje normalne wyjście (wyświetlanie tylko linii wskazanych w skrypcie komendą `p`)
`-e` wykonają podany skrypt (pojedyncze polecenie). Jeśli podajemy tylko jedną komendę ta opcja nie jest wymagana.
Składnia skryptu:
`[adres[,adres]] funkcja [argumenty]`
`adres` to numer linii pliku (`$` oznacza numer ostatniej linii) lub wyrażenie regularne umieszczone pomiędzy znakami `/`
. Określa on zakres linii strumienia na których będę dokonywane operacje. Na przykład `1,3` pasuje do pierwszych trzech linii, `/bash/` pasuje do wszystkich linii zawierających wyrażenie `bash`, zaś `/begin/,/end/` dotyczy wszystkich kolejnych linii z których pierwsza zawiera słowo `begin` a ostatnia słowo `end`. `funkcja` do wyboru mamy wiele możliwości edycji strumienia. Najważniejsze to:
`a tekst` dodaj podany tekst przed następną linią
`c tekst` zamień linię podanym tekstem
`d` usuń linię
`i tekst` wstaw podany tekst
`p` wyświetl bufor (aktualnie edytowaną linię)
`s/wyrażenie/łańcuch/` zastępuje podanym `łancuchem` pierwsze znalezione w buforze `wyrażenie`
`s/wyrażenie/łańcuch/g` zastępuje podanym `łancuchem` wszystkie znalezione w buforze `wyrażenia`
`=` wyświetla numer linii
Przykłady:
`> sed -n '1p' plik`
wyświetli pierwsza linię pliku
`> sed -n '3,$p' plik`
wyświetli wszystkie linie od 3-ciej to końca pliku
`> sed '3,$d' plik`
usunie wszystkie linie od 3-ciej do końca pliku
`> sed -n '/Marek/p' /etc/passwd`
wyświetla linie zawierające słowo `Marek` z pliku `/etc/passwd`
`> sed '/UNIX/c Linux' plik`
Zamienia linie w których występuje słowo `UNIX` zwrotem `Linux`
`> sed -n '/UNIX/=' plik`
wyświetla numery linii w których występuje wyrażenie `UNIX`
`> sed 's/UNIX/Linux/g plik`
zamienia wszystkie wystąpienia słowa `UNIX` na `Linux`
`> sed -n 's/UNIX/Linux/g plik`
tak jak wyżej ale wyświetlane są wyłącznie linie w których nastąpiła zmiana

grep

wyświetla linie pasujące do wzorca

Postać: `grep [opcje] wzorzec [plik...]`

Przykład:

```
> grep student /etc/passwd
```

wyświetli linie z pliku `/etc/passwd` zawierającą słowo tudent Często stosuje się to polecenie jako filtr w strumieniu, np:

```
> ls /bin | grep z | wc -l
```

wyświetli liczbę plików z katalogu `bin` zawierających w nazwie literę Najważniejsze opcje:
`-v` wyświetlane są wiersze w których wzorzec nie pojawia się
`-l` wyświetli tylko nazwę pliku w którym znaleziono wzorzec
`-i` nie rozróżnia dużych i małych liter we wzorcu
`-A n` wyświetla także n kolejnych linii
`-B n` wyświetla także n poprzedzających linii

Wyrażenia regularne (skrót)

Wybrane metaznaki wyrażeń rozszerzonych wyrażeń regularnych (POSIX ERE, *ang. Extended Regular Expressions*)
`[lista]` pasuje do pojedynczego znaku z danej listy
`[^lista]` pasuje do znaku nie podanego na liście
. (kropka) pasuje do dowolnego pojedynczego znaku
`\w` jest równoważne `[0-9a-zA-Z]` lub `[[:alnum:]]`, czyli zastępuje dowolna literę lub cyfrę
`\W` oznacza to samo co `$[^[[:alnum:]]]`
`^` i `$` to odpowiednio początek i koniec linii
`\<` oraz `\>` początek i koniec słowa
Po wyrażeniu regularnym mogą stać operatory powtórzenia:
`?` poprzedzający element pasuje zero lub jeden raz, np. `miark?a` pasuje do `miarka` ale też `miara`
`*` poprzedzający element pasuje zero lub więcej razy, np `W*in` pasuje zarówno do słowa `Windows` jak i do `Linux`
`+` poprzedzający element pasuje jeden lub więcej razy,
`{n}` poprzedzający element pasuje dokładnie `n` razy
`{m,n}` poprzedzający element pasuje od `m` do `n` razy
`|` operator LUB, np. `Fizyka|fizyka` pasuje do `fizyka` oraz `Fizyka`
`()` grupowanie, np. `fizy(ka|cy)` pasuje zarówno do `fizyka` i `fizycy`.

Uwaga: w podstawowych wyrażeniach regularnych (POSIX BRE *ang. Basic Regular Expressions*) stosowanych w większości narzędzi UNIXowych metaznaki `?`, `+`, `{}`, `()`, `|` tracą swoje szczególne znaczenie; zamiast nich należy użyć `\?`, `\+`, `\{\}`,`\(\)`, `\|`.

Przykłady:

```
grep 'bash$' /etc/passwd
```

linie zakończone słowem `bash` w pliku `/ert/passwd`

```
grep '\<[aA]' plik
```

linie w których występuje wyraz rozpoczynający się literą `a` lub `A`

`grep '^From: ' /var/mail/$USER`
lista odebranej poczty (linie rozpoczynające się słowem `From:`)
`grep -v '^$' plik`
wszystkie linie, które nie są puste
`grep '[0-9]\{9\}' plik`
dziewięciocyfrowe ciągi liczb, np. numery telefonów
`grep '(.\+) plik'`
psuje do dowolnego ciągu składającego się przynajmniej z jednego znaku zawartego w nawiasach

Inne przydatne polecenia i narzędzia (textutils): nano, emacs, vi, vim, awk, join, paste, tac, nl, od, split, csplit, uniq, comm, ptx, tsort, tr, fold

Potoki

`polecenie1 | polecenie2`
połączenie wyjścia programu 1 z wejściem programu 2

Przykłady:

```
> cat /etc/passwd | wc -l
> grep Marek /etc/passwd | cut -f 5 -d : | sort | head -n 1 > wybraniec
```

tee

czyta standardowe wejście i przesyła je na standardowe wyjście oraz do pliku.

Postać: `tee [-a] plik`

Najważniejsze opcje:
`-a` dopisuje zawartość strumienia wyjściowego do pliku (bez tej opcji zawartość pliku zostałaby nadpisana)
Przykład:

```
> grep Marek /etc/passwd | tee plik1.txt | wc -l
```

zapisze linie z pliku `/etc/passwd` zawierające słowo `marek` w pliku `plik1.txt`, zaś na ekranie wyświetlona zostanie ilość tych linii.