Instytut Fizyki
Uniwersytet Humanistyczno-Przyrodniczy
Kielce

Studia Informatyka
Rok I-szy. Języki i Środowisko Programisty.

ZAJĘCIA II – DOS – POLECENIA PLIKU WSADOWEGO

Polecenie ECHO – wypisuje podany tekst na ekranie;
Zapoznać się z komendą Echo/ ?
ECHO. – pozostawia wiersz pusty, ECHO bez kropki wyświetla aktualny stan Echa
Np. ECHO witaj studencie - spowoduje wyświetlenie na ekranie „witaj studencie"
Polecenie to umieszczone w pliku wsadowym spowoduje wyświetlenie dwóch wierszy:
ECHO witaj studencie
„witaj studencie"
Aby uniknąć wyświetlania tekstu polecenia należy wpisać ECHO OFF, przywraca polecenie ECHO ON.
ECHO OFF wstrzymuje wyświetlanie tekstu poleceń, łącznie z zachętą, ale nie ma wpływu na komunikaty, wywołane wykonywaniem poleceń. W przypadku pojedynczego wiersza ten sam wynik otrzymuje się poprzedzając polecenie znakiem @ (ale ECHO jest wyłączone tylko dla wiersza poprzedzonego tym znakiem np. @<polecenie> wypisujemy po to, żeby nie było widać jakie polecenie zawiera ten wiersz)
Np. polecenie COPY NOWYPLIK.TXT E: wywoła na ekranie
COPY NOWYPLIK.TXT A:
1 File(s) copied
podczas gdy
@COPY NOWYPLIK.TXT A: da wynik 1 File(s) copied (WIDOCZNE TYLKO SKUTKI)
ECHO jest w stanie wł

Przykład 1

```
copy con E:\test.bat
echo komunikat
^Z
Plików skopiowanych:        1

E:\>test
E:\>echo komunikat
komunikat
E:\>
copy con a:\test1.bat
@echo off
echo komunikat
^Z
Plików skopiowanych:        1
E:\>test1
komunikat
E:\>
```

Polecenie REM – jest komentarzem w pliku wsadowym. Komentarze służą jako dokumentacja programów, ale spowalniają ich wykonywanie. Nie będą one wyświetlone jeśli Echo jest wyłączone

Przykład 2

```
COPY CON KOMUNIKAT.BAT
@ECHO OFF
REM SPRAWDZANIE STANU DYSKIETKI
E:
CHKDSK E:/V
ECHO ON
CTRL+Z
```

Polecenie PAUSE – zatrzymuje wykonanie programu.
Jeśli ECHO jest włączone można je połączyć z dowolnym komunikatem, ale w tym wypadku zostanie wyświetlony także tekst polecenia. Lepiej korzystać z następującej formy polecenia w pliku wsadowym:

Przykład 3

```
COPY CON KOMUNIKAT1.BAT
@ECHO OFF
ECHO TO JEST KOMUNIKAT
PAUSE
ECHO ON
CTRL+Z
```

OBEJRZEĆ WYNIK - WYWOŁAĆ POLECENIE KOMUNIKAT1

Polecenie TYPE – umożliwia wyświetlenie bardziej skomplikowanych komunikatów, powyżej 122 znaków, np. podpowiedzi co do działania programu

4. Wykorzystanie zmiennych

Parametry (argumenty) zamienne
Pliki wsadowe mogą zawierać parametry, które zostaną zamienione na wartości podane w wierszu wywołania pliku. Plik może zawierać do dziesięciu parametrów: każdy z nich ma własną nazwę formalną w programie i może być zamieniony na odpowiednią wartość. Parametr zamienny oznaczony jest w pliku wsadowym cyframi 0-9, poprzedzonymi znakiem %. Pierwszy parametr %0 jest zastrzeżony dla nazwy samego pliku wsadowego. Następne odpowiadają parametrom rzeczywistym, o ile te ostatnie zostały podane, np.:

Przykład 4

```
COPY CON KOPIA.BAT
@ECHO OFF
COPY %1 E:\
COPY %2.TXT E:\
CTRL+Z
```

Utworzony został plik wsadowy przeznaczony do kopiowania plików na dysk E:
Polecenie wywoływane jest po wpisaniu KOPIA.

```
KOPIA        C:\WINDOWS\WMSETUP.LOG
C:\WINNT\WPLOG
```

Takie polecenie spowoduje zamianę pierwszego parametru %1 na plik wmsetup.log, a parametru %2 na wplog.txt

Przykład 5

```
COPY CON PRZYKŁAD1.BAT
@ECHO OFF
```

```
CD\
MD %1
CD %1
CHKDSK E:/%2 > %3
TYPE %3 |MORE
PAUSE
CD\
CD BAT
ECHO ON
CTRL+Z
```

WYNIK: PRZYKŁAD1 %1 %2 %3
PRZYKŁAD1 PIES V KOT
POLECENIE OBEJRZENIA DYSKIETKI I UTWORZONYCH KATALOGÓW

Polecenie SHIFT – pozwala na korzystanie z więcej niż 10 zmiennych, przez zastosowanie przesunięcia w lewo. Polecenie SHIFT może być stosowane w celu powtórzenia tych samych akcji w pętli kilka razy, z innymi parametrami za każdym razem.

Przykład 6

```
COPY CON TEST6.BAT
@ECHO OFF
ECHO %1 %2 %3
SHIFT
ECHO %1 %2 %3
SHIFT
ECHO %1 %2 %3
ECHO ON
CTRL+Z
```

Wywołanie polecenia TEST6 A B C da wynik
A B C
B C
C

1. ZMIENNE ŚRODOWISKA

Wartość zmiennych środowiska może być odzyskana poprzez podanie jej wartości z nawiasami %, %zmienna%
Zmienne środowiskowe już poznane: PATH, PROMPT, VER

PRZYKŁAD 1

```
COPY CON KOMUNIKAT.BAT
@ECHO OFF
ECHO DEFINICJA KURSORA ZACHĘTY JEST
NASTEPUJĄCA: %PROMPT%
ECHO ON
CTRL+Z
```

Można budować własne zmienne środowiskowe.
Definiowanie zaczyna się od komendy SET

PRZYKŁAD 2

```
COPY CON KOM2.BAT
@ECHO OFF
SET AGA=WŁÓŻ DYSKIETKĘ
ECHO %AGA%
PAUSE
SET AGA=        (bez wartości zostanie usunięta zmienna systemu)
ECHO ON
Ctrl+Z
```

Raz ustalona zmienna pozwala ją wywołać kilkakrotnie, dlatego warto ją zdefiniować %zmienna%. Można ją wywoływać w różnych programach. Gdy przestanie się z niej korzystać to trzeba ją wymazać polecenie SET=
Zmienne środowiskowe umieszczone są np. w pliku AUTOEXEC.BAT

2. PĘTLA FOR – służy w języku programowania do wielokrotnego powtarzania tej samej akcji, uruchamia polecenie dla każdego elementu z zestawu zawartego w liście

Składnia pętli:
FOR %%ZMIENNA IN (LISTA) DO POLECENIE
Gdzie poleceniem może być każde polecenie z DOSa, za wyjątkiem pętli, nie można umieszczać pętli w pętli (da się to zrobić w bardziej skomplikowanych językach programowania)
Występuje tu trzeci rodzaj zmiennej **%%zmienna** – przyjmuje wartości z listy, wykona polecenie z tą samą zmienną

PRZYKŁAD 4

```
COPY CON PTL1.BAT
@ECHO OFF
FOR %%X IN (1 2 3 4 5) DO ECHO %%X
CTRL+Z
```

PRZYKŁAD 4

```
COPY CON PTL2.BAT
@ECHO OFF
FOR  %%V IN (%1 %2 %3) DO COPY  %%V  A:\
CTRL+Z
```

W liście są parametry zamienne, wywołanie jej spowoduje, że będzie widać jakie są parametry zamienne.
Wywołanie polecenia: PTL2 E:\BAT\PTL1.BAT C:\CONFIG.SYS
Obejrzeć dysk E.

3. SKOKI I ETYKIETY:

GOTO – powoduje przejście do dowolnego wiersza pliku, nie do następnego
Etykieta – wiersz poleceń zaczynający się dwukropkiem, po którym następuje dowolna nazwa etykiety do 8 znaków, powinna się zaczynać literami
Makroinstrukcje interpretowane są wiersz po wierszu – nie ma możliwości, aby wykonał się np. wiersz pierwszy i ostatni. GOTO daje taka możliwość.

PRZYKŁAD 5 (Przykład złego programu, który nie ma możliwości skończenia się)

```
COPY CON PTL3.BAT
@ECHO OFF
```

: START *(to jest nazwa etykiety, do której wykonywany jest skok)- można wpisać polecenie REM*
COPY %1 E:
SHIFT
GOTO START *(wróci do STARTu)*
CTRL+Z

PTL3 (wpisać nazwę pliku)
Zatrzymanie: CTRL+C, CTRL+Scroll Look, CTRL+PAUSE, PAUSE

4. INSTRUKCJA WARUNKOWA:
Polecenie IF pozwala na przeprowadzenie testu w celu zdecydowania czy kolejne polecenie ma być wykonane czy ma być skok. Ma kilka postaci opartych o schemat:
IF WARUNEK POLECENIE
Polecenie jest wykonywane jeśli warunek jest spełniony lub
NOT IF WARUNEK POLECENIE
Polecenie jest wykonane gdy warunek nie jest spełniony.

PRZYKŁAD 6 (TEST RÓWNOŚCI)
IF [NOT] CIĄG1= =CIĄG2 POLECENIE
IF % = = ? GOTO POMOC
Co po podaniu znaku ? jako parametru spowoduje przejście do etykiety POMOC
Można zablokować np. dostęp do komputera poprzez ustalenie etykiety
COPY CON TEST_ROWNOSCI.BAT
@ECHO OFF
:START
IF %1= = XA29 GOTO POCZATEK
GOTO START
:POCZATEK
CTRL+Z

WYWOŁANIE: TEST_ROWNOSCI XA29

PRZYKŁAD 7 (TEST ISTNIENIA PLIKU)
Sprawdza czy istnieje plik i wykonuje polecenie gdy np. kopiujemy i nie chcemy nadpisywać jednego na drugi
IF [NOT] EXIST PLIK POLECENIE

COPY CON KOPIA2.BAT
@ECHO OFF
IF EXIST E:\%1 GOTO KOM
COPY %1 E:
GOTO KONIEC
: KOM
ECHO PLIK %1 JUŻ JEST NA DYSKIETCE A:
: KONIEC
ECHO ON
CTRL+Z

(bez skoku do KONIEC komunikat KOM byłby wyświetlany zawsze)
Wywołanie KOPIA2 nazwa pliku (ścieżka dostępu).

Przykłady zostały zaczerpnięte z książki:

D.Rougé, „MS-DOS pomysły i zastosowania", WNT, Warszawa 1992.

Instytut Fizyki
Uniwersytet Humanistyczno-Przyrodniczy
Kielce

Studia Informatyka
Rok I-szy. Języki i Środowisko Programisty.

ZAJĘCIA III – LINUX – WPROWADZENIE; OPERACJE NA PLIKACH I KATALOGACH.

1. Uruchom Linux-a w trybie graficznym, zaloguj się do systemu jako **student**
   a. Uruchom drugą konsolę (Alt+CTRL+F2), zaloguj się na niej jako **student**
2. Pracując na drugiej konsoli wywołaj polecenia:
   a. ls
   b. zapoznaj się z pomocą systemową wydając polecenia:**ls - -help** (oraz **ls –help|more**)
   c. man ls
   wyjście **:q**
3. Przejść na trzecią konsolę, utworzyć katalogi
   a. mkdir
      i. teksty
      ii. pisma
         (Polecenie **cd** pisma)
         1. pisma_przychodzace
         2. pisma_wychodzace
      iii. rysunki
      iv. test

Podczas tworzenia struktury katalogów przeglądaj ją poleceniem **ls – l**

4. Na drugiej konsoli przejdź na katalog główny „ / ” , poleceniem **ls** obejrzyj strukturę katalogów, a następnie skopiuj ją do pliku

   ls > przeglad_txt

5. Wyświetl zawartość utworzonego pliku wykorzystując polecenia **more** i **cat**

6.
   a. Utworzyć w kotalogu teksty dowiązanie miękkie do pliku przeglad_txt (polecenie **ln – s** istniejacy plik dowiazanie)
   b. Usuń katalog test
7. Poszukiwanie plików (polecenie **find**.). Korzystając z pomocy systemowej zapoznaj się z poleceniem find i jego najważniejszymi parametrami. Następnie wyszukaj w systemie plik przeglad_txt
8. Przejdź na konsolę 1 (Alt+F7). Sprawdź w trybie graficznym zmiany w strukturze katalogów systemu.
9. Uruchomić program Midnight Commander na obu konsolach
   a. (wywołanie **mc**)
   b. Zapoznać się z menu programu rozwijając linie poleceń.
10. W katalogu student utworzyć strukturę katalogów z poprzednich ćwiczeń wykorzystując **mc.**
11. Używając lewego lub prawego panelu **mc** zastosować różne tryby wyświetlania.
12. Używając lewego lub prawego panelu **mc** zastosować różne sposoby sortowania wyświetlanej informacji o katalogach i plikach.
13. W katalogu teksty utworzyć plik (**spis**) zawierający informację o przeglądanych katalogach.
14. Używając **mc** sprawdzić prawa dostępu do utworzonego pliku **spis,** następnie je zmienić (polecenie **chmod**) nadając mu wszystkie prawa użytkownika, grupy i pozostałych użytkowników.
15. Z menu POLECENIA wykonać:
    a. Drzewo katalogów
    b. Znajdź plik
    c. Historia poleceń
    d. Zadania w tle

Instytut Fizyki
Uniwersytet Humanistyczno-Przyrodniczy
Kielce

Studia Informatyka
Rok I-szy. Języki i Środowisko Programisty.

ZAJĘCIA IV – LINUX – SKRYPTY w POWŁOCE.

Wprowadzenie.

Często używane sekwencje poleceń można zapisać w pliku i poddać ten plik powłoce do interpretacji, zamiast wpisywać każdorazowo je z klawiatury. Pliki te nazywane są skryptami powłoki(shella).
Dobrym zwyczajem jest umieszczanie swoich skryptów w katalogu ~/bin.
Przykład 1
skrypt_1, podobny w działaniu do polecenia DIR z MS DOS:

```
$ cat > skrypt_1
pwd; ls -al
<ctrl-D>
$ _
```

Wykonanie takiego pliku jest możliwe z linii poleceń poprzez podanie:

```
$ bash skrypt_1
```

Po dodaniu atrybutów wykonywalności:

```
$ chmod +x skrypt_1
```

lub

```
$ chmod 755 skrypt_1
```

można wywoływać podany skrypt bezpośrednio:

```
$  ./skrypt_1
```

Pierwsza linia skryptu powinna wyglądać następująco:

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

Aby zorientować się, które pliki z zawartych w przeszukiwanych katalogach są skryptami shella należy wydać polecenie:

```
$ file * | grep Bourne
```

Wypisane wówczas zostaną wszystkie pliki zinterpretowane przez polecenie file jako skrypty shella Bournea.

Komentarze

Od znaku # do końca linii tekst jest uważany za komentarz.
Przykład 2
Użycie komentarza w skrypcie:

```
#
# program ilustrujący komentarze
#
echo "Mój pierwszy skrypt" # wyświetlenie linii
```

Przykład 3
Komentarz użyty bezpośrednio z poziomu shella:

```
$ ls -al |
> # inny sposób użycia komentarza
> more
```

Zmienne

Do odczytu wartości zmiennych ze standardowego wejścia służy polecenie read. Odczytuje ono linie ze standardowego wejścia, dzieli ją na słowa i przypisuje je do kolejnych zmiennych dostarczanych jako argumenty.
W bash dodatkowo jeśli po słowie read nie jest określona żadna zmienna, to odczytana wartość zostanie przypisana zmiennej REPLY.
Przykład 4
Wczytywanie danych w skrypcie skrypt_4:

```
echo "Wprowadź nazwy plików dla wejścia i wyjścia"
read plik1 plik2
echo Wejście: $plik1
echo Wyjście: $plik2
```

```
$ skrypt_4
Wprowadź nazwy plików dla wejścia i wyjścia
dane raport
Wejście: dane
Wyjście: raport
```

Przykład 5
Użycie zmiennej REPLY:

```
$ read
> moja zmienna
$ echo $REPLY
moja zmienna
```

Przekazywane parametrów do skryptu
Parametry pozycyjne służą do przekazywania argumentów z linii komend do skryptu (programu).
Bezpośrednio można zaadresować 9 parametrów, od $1 do $9, którym przypisane są argumenty
według schematu:

```
$ polecenie arg1 arg2 arg3 ...
     |      |    |    |
    $0     $1   $2   $3 ...
```

Dodatkowo bash pozwala na bezpośrednie zaadresowanie większej ilości parametrów, parametry dwucyfrowe należy podawać w postaci:

```
${nr-parametru}
```

Polecenie shift i set

Polecenie shift pozwala przesuwać parametry pozycyjne w lewo, tzn. pierwszy jest tracony,

drugi zostaje pierwszym, trzeci staje sie drugim, czwarty trzecim itd. Nie można przesuwać parametrów w prawo.
Do przypisania nowych wartości parametrom pozycyjnym służy polecenie set, które jest wykorzystywane również m.in. do odzyskiwania parametrów pozycyjnych, straconych po wykonaniu komendy shift.
Przykład 6
Oto skrypt skrypt_6, i jego użycie:

```
echo $1 $2 $3 $4
shift
echo $1 $2 $3 $4
shift 4
echo $1 $2 $3 $4
$ skrypt_6 a b c d e f g h i
a b c d
b c d e
f g h i
```

Przykład 7
Fragment skryptu analizującego parametry przekazywane do skryptu w linii poleceń:

```
while [ $# -ge 1 ]; do
case $1 in
zrób coś
esac
shift
done
```

Zmienne związane z linią poleceń.
Wartości niżej podanych zmiennych jest ustalana przez shell, nie może być ustalana przez użytkownika:
$0 - nazwa wykonywanego polecenia.
$# - liczba argumentów polecenia. Po wykonaniu polecenia shift, zmniejszana jest wartość
$#.
$* - wszystkie argumenty z linii polecenia w postaci jednego ciągu znaków.
$@ - wszystkie argumenty z linii polecenia w postaci osobnych ciągów znaków. Zmienna $@, jeśli nie występuje w cudzysłowie, jest identyczna jak zmienna $*.
$$ - numer aktualnie wykonywanego procesu (PID), czyli numer PID skryptu, zmienna ta często jest używana do tworzenia plików tymczasowych, np. tmp=/tmp/moj.$$;
$! - numer procesu (PID) ostatniej komendy wykonywanej w tle;
$? - status wyjścia (exit status) ostatnio wykonywanej komendy;

Dobrze napisany skrypt weryfikuje liczbę parametrów z jaką jest wywoływany, oto kilka przykładów:
Przykład 8
Skrypt data, który powinien być wywołany w sposób następujący:

```
$ data 19 02 2000
if [ $# -ne 3 ]; then
echo 1>&2 Wywołanie: $0 dd mm rrrr
exit 127
fi
```

Przykład 9
Rozważmy następujący skrypt skrypt_9:

```
pos=$*; echo $pos
num=$#; echo $num
shift
echo $#
$ skrypt_9 a b c
a b c
3
2
123
```

Eksportowanie zmiennych
Eksportowanie zmiennych wykonuje się używając polecenia export:

```
$ export zmienna
```

Przykład 10
skrypt_10:

```
user3=C
echo x = $x
echo user1 = $user1 " " user2 = $user2 " " user3
= $user3

$ user1=A; user2=B; skrypt_10
$ x=local
$ user1=A user2=B skrypt_10
x=
user1 = A user2 = B user3 = C
$ echo user1=$user1 user2=$user2
user1= user2=
```

Status wyjścia (exit status)
W skryptach shella przyjęte jest, że jeśli program kończy się sukcesem wówczas do procesu wywołującego zwracana jest wartość 0, w przeciwnym wypadku zwracana jest wartość różna od zera.
Status wyjścia można odczytać korzystając z parametrów $? i $!
Przykład 11

```
$ ls
bin Mail
```

```
$ echo $?
0
$ ls -z
ls: illegal option -z
$ echo $?
2
```

Polecenie exit, exit [n]
Wyjście ze skryptu ze statusem n (np. exit 1). Jeżeli n nie jest podane, status wyjściowy będzie taki, jak ostatnio wykonanego polecenia.
Przykład 12

```
skrypt_12:
echo Przykład działania komendy exit
exit 127
$ skrypt_12
Przykład działania komendy exit
$ echo $?
127
```

Polecenia true i false
Oba polecenia generują status wyjścia, przy czym true generuje zawsze status wyjścia 0, a false generuje zawsze status wyjścia 1.
Przykład 13

```
$ if true
> then
> date
> fi
Tue Oct 23 11:20:31 EDT 1990
```

Polecenie test
Służy do testowania wyrażeń numerycznych, plikowych i tekstowych, zwraca 0 (prawda) lub wartość różną od zera (fałsz).
test wyrażenie lub [wyrażenie]
Należy pamiętać, aby wpisywać spacje po obu stronach wyrażenia, używać cudzysłowów jeśli używa się dodatkowych spacji w wyrażeniu. Możliwe jest grupowanie komend w nawiasach objętych cudzysłowami.
Operatory instrukcji test:
! not
-a and
-o or
Testowanie typów plików:
-s plik plik istnieje i jest niezerowy;
-f plik plik istnieje i jest zwykłym plikiem;
-d plik plik istnieje i jest katalogiem;
-r plik plik istnieje, a wywołujący użytkownik posiada prawo czytania;
-w plik plik istnieje, a wywołujący użytkownik posiada prawo zapisu;
-x plik plik istnieje, a wywołujący użytkownik posiada prawo wykonywania;
p1 -nt p2 plik p1 jest nowszy niz plik p2;
p1 -ot p2 plik p1 jest starszy niz plik p2.
Testowanie wyrażeń numerycznych:
w1 -eq w2 w1 = w2;
w1 -ge w2 w1 > w2;
w1 -gt w2 w1 > w2;
w1 -le w2 w1 6 w2;
w1 -lt w2 w1 > w2;
w1 -ne w2 w1 6= w2.
Testowanie wyrażeń tekstowych:
te prawda gdy tekst te jest nie zerowy;
-z te prawda gdy tekst te jest zerowej długości;
-n te prawda gdy tekst te jest dłuższy od zera;
te1 = te2 prawda gdy teksty te1 i te2 są identyczne;
te1 != te2 prawda gdy teksty te1 i te2 nie sa identyczne;
Przykład 14

```
if test -d $plik
then
echo $plik jest katalogiem
else
echo $plik jest plikiem
fi
if [ $Rok -lt 1901 -o $Rok -gt 2099 ]; then
echo 1>&2 Rok \"$Rok\" jest poza zakresem.
exit 127
fi
test -w /home/kowalski -a -d /home/jan
[ -w plik1 -o ! -x plik2 ]
[ ! \( -w plik1 -a -x plik1 \) ]
```

Konstrukcje programowe
Instrukcje warunkowe
Instrukcja if

```
if warunek
then
polecenia
fi
        lub
if warunek
then
polecenia
else
polecenia
fi
```

Warunek może być formułowany z jednego lub kilku poleceń, przy czym if testuje tylko status wyjścia ostatniego polecenia.

Konstrukcja elif umożliwia łatwiejsze i szybsze budowanie zagnieżdżonych konstrukcji if.

Zagnieżdżone instrukcje if — konstrukcja elif

```
if warunek
then
polecenia
else
if warunek
then
polecenia
else
polecenia
fi
fi
```

```
if warunek
then
polecenia
elif warunek
then
polecenia
else
polecenia
fi
```

Skrócona wersja instrukcji if — wykonywanie sekwencyjne

Jest to mechanizm umożliwiający wykonanie następnego polecenia, w zależności od wyniku zakończenia polecenia poprzedniego.

polecenie1 &&polecenie2&&polecenie3 &&...
polecenie1 ||polecenie2||polecenie3 ||...

Przykład 15

Polecenie: who|grep user1 > /dev/null && echo "UWAGA user1 pracuje!"
jest równoważne:

```
if who|grep user1 > /dev/null
then
echo "UWAGA user1 pracuje!"
fi
```

polecenie: who|grep user1 > /dev/null || echo "user1 nie pracuje!"
jest równoważne:

```
if who|grep user1 > /dev/null
then
...
else
echo "user1 nie pracuje!"
fi
```

Konstrukcja case

```
case warunek in
wzorzec1 )
polecenia
;;
wzorzec2 )
polecenia
;;
*)
polecenia
;;
esac
```

Konstrukcja case używana jest alternatywnie do mnogich if. Umożliwia wybór jednego z podanych wzorców w celu wykonania określonej grupy poleceń.
We wzorcach mogą występować metaznaki. W jednym wzorcu można umieścić kilka wartości oddzielając je znakami |, np. Y|y|t|T. Metaznak * zastępuje wszystkie wzorce, oznacza pozostałe przypadki, jeśli jest zawarty w instrukcji case musi być podany jako ostatni.

Przykład 16

```
case "$Rok" in
[0-9][0-9])
Rok=19${Rok}
;;
[0-9][0-9][0-9][0-9])
;;
*)
esac
```

Pętle

Pętla for

```
for zmienna in arg1 arg2 ...
do
polecenia
done
```

Konstrukcja ta wykonuje polecenia kolejno dla każdego argumentu podanego w liście argumentów

Przykład 17

```
#usuwania plików pośrednich
for i in /tmp /usr/tmp
do
rm -rf $i/*
done
```

Przykład 18

```
#zmiana praw dostepu
for i in `ls`
do
chmod 740 $i
done
```

Petle while i until

```
until warunek
do
polecenia
done
```

```
while warunek
do
polecenia
done
```

Pętla while wykonuje polecenia tak długo, aż testowany status wyjścia warunku (lub ostatniego polecenia, jeśli warunek jest złożony) będzie 0. Przeciwieństwem pętli while jest pętla until, która kontynuuje wykonywanie poleceń tak długo, aż testowany status wyjścia warunku

będzie różny od zera. Należy pamiętać, iż warunek może być złożony z wielu poleceń, wszystkie one będą wykonane co najmniej raz, a tylko status wyjścia ostatniego z nich wpływa na wykonanie poleceń z bloku do-done.

Instrukcja break

break przerywa wykonanie instrukcji pętli i powoduje przekazanie sterowania do pierwszej instrukcji
poza pętla.
Przykład 18
Przykład programu zawierającego instrukcje break, oraz program alternatywny, nie zawierający tej instrukcji:

```
while true
do
polecenie1
if [ -f plik1 ]; then
rm plik1
else
break
fi
polecenie2
done
polecenie3

polecenie1
while [ -f plik1 ]
do
rm plik1
polecenie2
polecenie1
done
polecenie3
```

Instrukcja continue

continue powoduje przerwanie sekwencyjnego wykonywania instrukcji wewnątrz pętli i przejście na początek pętli.
Przykład 19
Przykład programu zawierającego instrukcje continue, oraz program alternatywny, nie zawierający tej instrukcji:

```
for var in $*
do
if [ $var -ge 0 ]; then
arg=$arg" "$var
continue
fi
var=‘expr $var\*-1‘
arg=$arg" "$var
done
echo $arg

for var in $*
do
if [ $var -lt 0 ]; then
var=‘expr $var\*-1‘
fi
arg=$arg" "$var
done
echo $arg
```

Funkcje w skryptach

Powłoka pozwala definiować nie tylko zmienne, ale i funkcje, które mogą zawierać dowolny ciąg poleceń. Funkcje mogą być wywoływane z argumentami, z taka sama składnia jaka obowiązuje przy przekazywaniu parametrów do skryptu.

```
nazwa_funkcji ()
{
polecenia
}
```

Przykład 20
Przykład funkcji wczytaj:

```
wczytaj ()
{
echo -n ${1:-’Podaj nazwe: ’}
read nazwa
echo Wybrano ${nazwa:=$2}
}
```

Reagowanie na sygnały zewnętrzne - polecenie trap

Poruszona teraz zostanie kwestia reakcji wykonywanego skryptu na działania zewnętrzne. W systemie UNIX użytkownik może wysłać procesowi sygnał np. sygnał przerwania. Można wymagać od skryptu, aby na taki sygnał reagował w określony sposób, np. wyświetlając komunikat. Do obsługi mechanizmu sygnałów w skryptach służy polecenie trap, o składni:
trap [polecenie][numer sygnału]
Oba parametry są opcjonalne.
Wysłanie dowolnego sygnału do procesu można uzyskać za pomocą polecenia kill:

kill -nr PID

gdzie nr oznacza numer wysyłanego sygnału, a PID identyfikuje proces do jakiego sygnał ma zostać wysłany.

Instytut Fizyki
Uniwersytet Humanistyczno-Przyrodniczy
Kielce

Studia Informatyka
Rok I-szy. Języki i Środowisko Programisty.

Zadania:

1. Napisz skrypt przyjmujący trzy parametry: nazwę pliku, rozszerzenie i opcję (-k lub –p). Skrypt powinien, w zależności od podanej opcji, odpowiednio skopiować dany plik lub przenieść pod nazwę postaci *stara_nazwa.rozszerzenie*.
2. Napisz skrypt wypisujący wszystkie trzyelementowe wariacje bez powtórzeń zbioru czteroelementowego.

Instytut Fizyki
Uniwersytet Humanistyczno-Przyrodniczy
Kielce

Studia Informatyka
Rok I-szy. Języki i Środowisko Programisty.

ZAJĘCIA V – KOMPILACJA w gcc

Używanie GNU cc (gcc)

gcc daje programiście dużą kontrolę nad procesem kompilacji. Proces kompilacji składa się z do czterech etapów: *przetwarzanie wstępne, kompilacja, asemblacja* oraz *łączenie.* Możemy zatrzymać ten proces na dowolnym etapie, aby zbadać wyjście z kompilatora. gcc może obsługiwać różne dialekty C, takie jak ANSI C lub tradycyjne C (Kernighan i Ritchie). Można kontrolować ilość i typ informacji wyszukiwania błędów, włączanych do wynikowych kodów binarnych, ponadto, podobnie jak większość kompilatorów, gcc może również optymalizować kod.

gcc zawiera ponad 30 różnych ostrzeżeń oraz trzy poziomy ostrzeżeń typu „złap wszystko". Jest również *kompilatorem skrośnym* (ang. *cross-compiler),* więc na jednej architekturze procesora możemy tworzyć kod, który będzie uruchamiany na innej. Kompilowanie skrośne jest ważne, ponieważ Linux działa na wielu różnych systemach, takich jak Intel x86, PowerPC, Amiga oraz Sun Sparc. Każdy procesor ma inną architekturę, więc sposób, w jaki kody binarne powinny być konstruowane jest inny dla każdego systemu.

Wywoływanie gcc

Aby użyć gcc, należy podać nazwę pliku z kodem źródłowym C i określić nazwę pliku wyjściowego, używając opcji -o. gcc wykona przetwarzanie wstępne, skompiluje kod, wykona asemblację i łączenie programu, tworząc plik wykonywalny, często nazywany również *binariami.* Oto prosta składnia:

gcc infile.c [-o outfile]

infile.c jest nazwą pliku z kodem źródłowym w C, a opcja -o mówi, że nazwą pliku wyjściowego jest outfile. Nawiasy kwadratowe [ ] wskazują, że argument jest opcjonalny (nieobowiązkowy). Jeżeli nazwa pliku wyjściowego nie jest określona, gcc nadaje
plikowi wyjściowemu domyślnie nazwę a. out.

Poniższy przykład używa gcc do tworzenia programu hello z pliku źródłowego hello. c. Najpierw kod źródłowy:

PRZYKŁAD_1

```
/* Listing 1.1
 * hello.c - klasyczny pogram "Hello, world.1 "
 */
#include <stdio.h>
int main(void)
{
puts( "Hello, świat programowania Linuksa! ");
return 0;
}
```

Aby skompilować i uruchomić ten program, wpiszmy:

```
$ gcc hello.c -o hello
```

Jeżeli wszystko pójdzie dobrze, gcc wykona swoja pracę po cichu i powróci do znaku gotowości powłoki, gcc kompiluje i łączy plik źródłowy hello. c, tworząc binaria, określone przez argument opcji -o, hello.

Polecenie ls pokazuje, wykonuje program, \

WYNIKI

```
$ ls
hello.c     hello*
$ ./hello
Hello, świat programowania Linuksa!
```

Wywoływanie gcc krok po kroku

W pierwszym przykładzie wiele operacji odbywało się w sposób niewidoczny. Aby rozwinąć makra i wstawić zawartość plików #included, gcc przepuścił hello.c przez preprocesor, cpp. Następnie skompilował wstępny kod źródłowy na kod obiektowy. Na koniec program scalający (ang. *linker),* id, utworzył kod binarny hello.

Kroki te możemy odtworzyć ręcznie, przechodząc po kolei przez proces kompilacji. Aby powiedzieć gcc, aby zatrzymał kompilację po wstępnym przetworzeniu, użyjmy opcji -E, tak jak poniżej:

```
$ gcc -E infile.c -o outfile.cpp
```

Następnym krokiem jest kompilacja wstępnie przetworzonego pliku na kod obiektowy, za pomocą opcji c. Prawidłowa składnia dla tego kroku jest następująca:

```
$ gcc -x infile-type -c infile.cpp -o outfile.o
```

W końcu łączenie pliku obiektowego tworzy obraz binarny. Polecenie łączenia powinno wyglądać mniejwięcej tak:

```
$ gcc infile.o -o outfile
```

PRZYKŁAD_2

Powróćmy do programu hello z poprzedniego przykładu i przejdźmy krok po kroku przez proces kompilacji, tak jak ilustruje to poniższy przykład. Najpierw przetwórzmy wstępnie hello. c:

$ gcc -E hello.c -o hello.cpp

Jeżeli za pomocą edytora tekstu zajrzymy do pliku hello.cpp, zobaczymy, że zawartość pliku stdio.h rzeczywiście została włączona do pliku wraz z innymi wstępnie przetworzonymi symbolami. Poniższy listing jest fragmentem pliku hello. cpp:

```
extern int fgetpos (FILE *  stream, fpos_t  *_pos) ;
extern int fsetpos   (FILE *__stream, const fpos_t *__pos) ;
# 519 "/usr/include/stdio.h" 3
# 529 "/usr/include/stdio.h" 3
extern void clearerr (FILE *__stream) ;
extern int feof (FILE *__stream) ;
extern int ferror (FILE *__stream)    ;
extern void clearer_unlocked (FILE *__stream) ;
extern int feof_unlocked (FILE *__stream)    ;
extern int ferror_unocked (FILE *__stream)    ;
extern void perror (__const char *__s)    ;
extern int sys_nerr ;
extern __const char *__const sys__errlist[ ] ;
```

Następnym krokiem jest kompilacja hello. cpp na kod obiektowy:

$ gcc -x cpp-output -c hello.cpp -o hello.o

Aby gcc rozpoczął kompilację w określonym momencie, w tym przypadku od wstępnie przetworzonego kodu źródłowego, musimy użyć opcji -x. Na koniec łączenie pliku obiektowego tworzy plik binarny:

$ gcc hello.o -o hello

Na szczęście, łatwiejsze jest użycie „skróconej" składni: gcc hello. c -o hello. Poniższy przykład pokazuje, że możemy zatrzymać i rozpocząć kompilację na dowolnym etapie, jeżeli powstanie taka potrzeba.

Używanie wielu plików źródłowych

Większość programów C składa się z wielu plików źródłowych, więc przed końcowym krokiem łączenia każdy z plików tych musi być skompilowany na kod obiektowy. Nie jest to trudne. Należy po prostu w wierszu poleceń gcc podać nazwę wszystkich plików źródłowych, które mają być skompilowane. Reszta zostanie wykonana automatycznie. Wywołanie gcc będzie wyglądać mniej więcej tak:

$ gcc filel.c file2.c file3.c -o progname

gcc najpierw tworzy filel.o, file2.o i file3.o , a następnie łączy je, aby utworzyć progname.

PRZYKŁAD_3

Załóżmy, że pracujemy nad programem newhello, który używa kodu showit.c i msg.c:

showit.c

```
/*
* *showit.c program obsługi ekranu
*/
#include <stdio.h>
#include "msg.h"
 int main(void)
{
char msg_hi[ ] = {"Hi there, programmer!"} ;
char msg_bye[ ] = {"Goodbye,
programmer!"};
printf ("%s\n", msg_hi);
prmsg(msg_bye) ;
return 0;
}
```

msg.h

```
/*
*msg.h plik nagłówkowy dla msg.c
 */
#ifndef MSG_H_
#define MSG_H_
void prmsg(char *msg) ;
#endif /* MSG_H_ */
```

msg.c

```
/*
msg.c definicje funkcji z msg.h
 */
#include <stdio.h>
#include "msg.h"
void prmsg(char *msg)
{
printf("%s\n", msg);
}
```

Poleceniem kompilującym te programy, a więc i tworzącym newhello, jest:

$ gcc msg.c showit.c -o newhello

Oto rezultat uruchomienia programu:

$ ./newheilo
Hi there, programmer!
Goodbye, programmer!

Opcje i argumenty

Lista opcji wiersza poleceń przyjmowanych przez gcc zajmuje kilkanaście stron, tak że tabela zawiera tylko najczęściej używane. Tabela . Opcje wiersza poleceń gcc

Opcja	Zastosowanie
	Nazwa pliku wyjściowego plik (nie jest konieczna przy kompilacji kodu obiektowego). Jeżeli nazwa pliku nie jest określona, domyślną nazwą jest a. out.
-c	Kompilacja bez scalania.
-Dfoo=bar	Definiuje w wierszu poleceń makro preprocesora foo z wartością ba r.
-Inazwakatalogu	Umieszcza katalog określony w nazwakatalogu na początku listy katalogów, które gcc przeszukuje w poszukiwaniu plików nagłówkowych.
-Lnazwakatalogu	Umieszcza katalog określony w nazwakatalogu na początku listy katalogów, które gcc przeszukuje w poszukiwaniu plików biblioteki. Domyślnie, gcc łączy z bibliotekami dzielonymi.
-static	Scala z bibliotekami statycznymi.
-lf oo	Scala z biblioteką foo.
-g	Wstawia standardowe informacje wyszukiwania błędów do pliku binarnego.
-ggdb	Wstawia mnóstwo informacji wyszukiwania błędów do pliku binarnego. Informacje te rozumie debuger GNU, gdb.
-O	Optymalizuje kompilowany kod. Opcja ta jest równoważna opcji -01.
-On	Określa poziom optymalizacji n, 0<=n<=3.
-ansi	Przełącza kompilator w tryb ANSI/ISO C, nie pozwalając na rozszerzenia GNU, które są sprzeczne ze standardem.
-pedantic	Wyświetla wszystkie ostrzeżenia wymagane prze standard ANSI/ISO C.
-pedantic-errors	Wyświetla wszystkie błędy wymagane przez standard ANSI/ISO C.
-traditional	Włącza obsługę składni C Kemighan i Ritchie (jeżeli nie rozumiemy, co to znaczy, nie przejmujmy się tym).
-W	Wyłącza wszystkie komunikaty ostrzeżeń.
-Wali	Wyświetla wszystkie ogólnie akceptowalne ostrzeżenia dostarczane przez gcc. Aby zobaczyć określone ostrzeżenia, używamy -Wwarning.
-werror	Zamiast generować ostrzeżenia, gcc przekształca ostrzeżenie w błędy, zatrzymując kompilację.
-MM	Daje na wyjściu kompatybilną z mąkę listę zależności. Opcja ta jest użyteczna przy tworzeniu pliku mąkę w skomplikowanym projekcie.
-v	Pokazuje polecenia użyte w każdym z kroków kompilacji.

Sprawdzenie błędów i ostrzeżenia

gcc oferuje całą klasę opcji wiersza poleceń służących do sprawdzania błędów i generowania ostrzeżeń. Opcje te obejmują -ansi, -pedantic, -pedantic-errors i -Wali. Rozpocznijmy od -pedantic. Opcja ta powoduje, że gcc generuje wszystkie ostrzeżenia wymagane przez ścisły standard C ANSI/ISO. Wszystkie programy używające niedozwolonych rozszerzeń, takich jak rozszerzenia wspierane przez gcc, będą odrzucone. Opcja -pedantic-errors zachowuje się podobnie, z tą różnicą, że zamiast ostrzeżeń generowane są błędy. Opcja -ansi wyłącza rozszerzenia GNU, które nie są zgodne ze standardem. Jednak żadna z tych opcji nie gwarantuje, że kod, nawet jeżeli skompiluje się bez błędów z którąś z tych opcji, jest w 100% zgodny z ANSI/ISO.

Opcja -Wali instruuje gcc, aby wyświetlać wszystkie ostrzeżenia, które mogą odnosić się do kompilowanego kodu. Opcja -Wali generuje ostrzeżenia, które większość programistów uważa za wątpliwe i łatwe do zmodyfikowania, aby zapobiec ich generowaniu. Jednym z przykładów złej praktyki kodowania jest zadeklarowanie zmiennej, która potem nie zostanie nigdzie użyta. Innym przykładem jest zadeklarowanie zmiennej bez określenia wprost jej typu.

PRZYKŁAD_4

Rozważmy poniższy przykład, który pokazuje bardzo zły styl programowania. Zdeklarowana jest tu funkcja main jako zwracająca void, podczas gdy w rzeczywistości funkcja main zwraca int. ponadto użyte jest rozszerzenie GNU long long int, deklarujące liczbę całkowitą 64-bitową.

pedant.c

```
/*
*pedant.c - użyj -ansi, -pedantic lub -pedantic-errors
*/ #include <stdio.h>
```

```
void main(void)
{
long long int i = 01;
puts("This is a non-conforming C program");
}
```

1. Najpierw spróbuj skompilować powyższy kod bez sprawdzania zgodności:

```
$ gcc pedant.c –o pedant
pedant.c: In function 'main':
pedant.c:8: warning: return type of 'main' is not 'int'
```

2. Następnie wykonaj kompilację za pomocą opcji –ansi:

```
$ gcc –ansi pedant.c –o pedant
pedant.c: In function 'main':
pedant.c:8: warning: return type of 'main' is not 'int'
```

Opcja –ansi zmusza gcc do wygenerowania komunikatu diagnostycznego wymaganego przez standard. Nie zapewnia ona jednak, że kod jest zgodny z ANSI C. Program został skompilowany, pomimo nieprawidłowej deklaracji funkcji main.

3. Użyj opcji –pedantic:

```
$ gcc –pedantic pedant.c –o pedant
pedant.c: In function 'main':
pedant.c:9: warning: ANSI C does not support 'long long'
pedant.c:8: warning: return type of 'main' is not 'int'
```

Podobnie kod jest kompilowany, pomimo wygenerowanych ostrzeżeń.

4. Na koniec użyj opcji –pedantic-errors:

```
$ gcc –pedantic-errors pedantic.c –o pedant
pedant.c: In function 'main':
pedant.c:9: ANSI C does not support 'long long'
pedant.c:8: return type of 'main' is not 'int'
```

Program tym razem nie jest skompilowany. gcc kończy kompilację po wyświetleniu wymaganych błędów diagnostycznych.

Opcje debugowania.

Użycie opcji –g i –ggdb spowoduje wstawienie informacji wyszukiwania błędów do kompilowanych programów.

Opcja -g i -ggdb jest używana z wartościami 1, 2 lub 3, określającymi, jak wiele informacji debugowania należy wstawić do programu.

PRZYKŁAD_5

Przykład poniższy pokazuje, w jaki sposób należy skompilować program z informacjami wyszukiwania błędów oraz jaki wpływ mają symbole debugowania na wielkość plików binarnych. Rozmiar pliku po kompilacji może być zaskoczeniem.

```
$ gcc –g helloc.c –o hello_g
$ ls –l  hello_g
-rwxr—xr-x  …   ...hello_g*
$ gcc –ggdb hello.c –o hello_ggdb
$ ls –l  hello_ggdb
-rwxr-xr-x  …   … hello_ggdb
```

Instytut Fizyki
Uniwersytet Humanistyczno-Przyrodniczy
Kielce

Studia Informatyka
Rok I-szy. Języki i Środowisko Programisty.

ZAJĘCIA VI – KOMPILACJA – make

make

jest narzędziem stosowanym do sterowania procesem budowania i ponownego budowania oprogramowania, make automatyzuje czynności związane z budową, sposobem i czasem budowania. Dzięki temu programista może skoncentrować się na pisaniu kodu. Oszczędza to również programiście wiele wpisywania, ponieważ zawiera kompilator C wywoływany z odpowiednimi opcjami i argumentami. Zadania składające się z wielu plików źródłowych wymagają skomplikowanych wywołań kompilatora, make upraszcza to, umieszczając te polecenia w *pliku make* - pliku tekstowym zawierającym wszystkie polecenia wymagane do zbudowania programu.

Tworzenie plików make

W jaki sposób więc make osiąga opisane wyżej magiczne rezultaty? Wykorzystuje do tego celu plik make. *Plik make* jest plikiem tekstowym bazy danych, zawierającym reguły, które mówią make, co budować i w jaki sposób. *Reguła* składa się z następujących elementów:

" *Cel,* czyli to, co make ostatecznie próbuje utworzyć.
" Lista jednej lub więcej *zależności, zazwyczaj* plików, wymaganych do zbudowania celu.
" *Lista poleceń* do wykonania, aby utworzyć cel z określonych zależności.

Gdy GNU make zostanie wywołany, szuka pliku o nazwie GNUmakefile, makefile lub Makefile, właśnie w takiej kolejności. Z pewnych powodów większość programistów Linuksa używa ostatniej formy Makefile.
Reguły pliku make mają ogólną formę

```
cel : zależność [zależność] [...]
polecenie
[polecenie]
[...]
```

Wywoływanie make

Wywoływanie make jest bardzo proste i sprowadza się do wydania polecenia make w katalogu zawierającym plik make. Oczywiście, podobnie jak większość programów GNU, make oferuje mnóstwo opcji wiersza poleceń. Najczęściej używane opcje są przedstawione w tabeli

Opcja	Przeznaczenie
-f plik	Użyty zostanie plik make o nazwie plik, zamiast pliku o standardowej nazwie (GNUmakefile, makefile lub Makefile)
-n	Wyświetlane (lecz nie wykonywane) są polecenia, które byłyby wykonane przez make. Opcja ta jest użyteczna przy testowaniu pliku make
-Inazwakatalogu	Dodaj nazwakatalogu do listy katalogów, które make powinien przeszukać w poszukiwaniu wstawionych plików make
-s	Opcja ta powoduje, że make nie wypisuje wykonywanych poleceń
-w	Wyświetlane są nazwy katalogów, podczas zmienia katalogów przez make
-Wplik	Wykonaj make, tak jakby plik *plik* był zmodyfikowany. Podobnie jak opcja -n, opcja ta jest bardzo użyteczna przy testowaniu plików make
-r	Wyłącz wszystkie wbudowane reguły make.
-d	Opcja ta powoduje wyświetlenie mnóstwa informacji debugowania.
-i	Normalnie make zatrzymuje się, jeżeli polecenie zwraca niezerowy kod błędu. Opcja ta wyłącza to zachowanie.
-k	Przy tej opcji make kontynuuje działanie, nawet jeżeli jeden z celów nie zostanie zbudowany. Normalnie make kończy działanie, jeśli jeden z celów nie zostanie zbudowany.
-jN	Uruchamiane jest N poleceń równolegle, gdzie N jest małą, niezerową liczbą całkowitą.

Przykłady

Wszystkie przykłady używają poniższego pliku make.

```
#
#  Makefile
#
LDFLAGS :=
# Przestaw znak komentarza w dwóch poniższych
#wierszach jeśli chcesz włączyć optymalizację
CFIAGS := -g $(CPPFLAGS)
#CFIAGS := -02 $(CPPFLAGS)
PROGS = \
hello \
pedant \ newhello \
newhello-lib \
noret \
inline \
packme
all: $(PROGS)
.c.o:
$(CC) $(CFLAGS) -c $*.c
.SUFFIXES: -c .o
hello: hello.c
pedant: pedant.c
newhello: showit.c msg.c
$(CC) $(CFLAGS) $^ -o $@
newhello-lib: showit.c libmsg.a
$(CC) $(CFIAGS) $< -o $@ -I. -L. -lmsg
libmsg.a: msg.c
$(CC) $(CFLAGS) -c $< -o libmsg.o
$(AR) rcs libmsg.a libmsg.o
noret: noret.c
```

Instytut Fizyki
Uniwersytet Humanistyczno-Przyrodniczy
Kielce


```
inline: inline.c
$ (CC) -02 $< -o $@
paciane: packme. c
.PHONY : clean zip
clean:
$(RM) $(PROGS) *.o *.a *~ *.out
zip: clean
zip 215101cd.zip *.c *.h Makefile
```

Pierwszy przykład ilustruje, w jaki sposób za pomocą opcji -w i -n symulować budowanie.

```
$ make -Wmsg.c -Wshowit.c -n newhello-lib
cc -g -c msg.c -o libmsg.o
ar rcs libmsg.a libmsg.o
cc -g showit.c -o newhello-lib -I. -L. -lmsg
```

Opcja -Wplik powoduje, że make zachowuje się tak, jakby plik został zmieniony, natomiast opcja -n wyświetla polecenia, które byłyby wykonane, bez ich wykonywania. Jak widać, make najpierw zbudowałby plik biblioteczny, libmsg.a, następnie skompilował showit.c i wykonał łączenie z biblioteką, aby utworzyć newhello-lib. Jest to doskonały sposób na testowanie reguł pliku make.

Drugi przykład Ten przykład porównuje normalne wyjście z make z rezultatem uzyskanym przy użyciu opcji -s, ograniczającej informacje wyświetlane przez mąkę.

```
$ make all
cc -g     hello.c   -o hello
cc -g     pedant.c   -o pedant
cc -g showit.c msg.c -o newhello
cc -g -c msg.c -o libmsg.o
ar rcs libmsg.a libmsg.o
pedant, c: In function 'main':
pedant.c:8: warning: return type of main' is not 'int'
cc -g     noret.c   -o noret
cc -o2 inline.c -o inline
cc -g     packme.c   -o packme
cc -g showit.c -o newhello-lib -I. -L. -lmsg
$ make -s all
pedant, c: In function 'main':
pedant.c:8: warning: return type of main' is not 'int'
```

Opcja -s wyłącza wszystkie informacje wyświetlane przez make. Ostrzeżenie na temat wartości zwracanej przez main jest komunikatem diagnostycznym gcc, na co oczywiście make nie ma wpływu.

Literatura:

Kurt Wall, LINUX programowanie w przykładach, MIKOM, Warszawa, 2000

Następnie proszę otworzyć stronę www:
http://wazniak.mimuw.edu.pl/index.php?title=%C5%9Arodowisko_programisty/Automatyzacja_kompilacji_-_make
przestudiować ją i wykonać wszystkie znajdujące się tam przykłady.

Instytut Fizyki
Uniwersytet Humanistyczno-Przyrodniczy
Kielce

Studia Informatyka
Rok I-szy. Języki i Środowisko Programisty.

ZAJĘCIA – PROGRAMOWANIE C - PODSTAWY .

Formatowane wyjście i wejście

1.
```
#include <stdio.h>

int main()
{
 printf("Witaj, studencie\n");
 return 0;
}
```
2.
```
#include <stdio.h>

int main()
{
 printf("Witaj, ");
 printf("studencie");
 printf("\n");
 return 0;
}
```
3.
```
/*
dodatkowe znaki sterujace: \a, \b, \f, \n, \r, \t, \\, \?, \', \"
*/
#include <stdio.h>

int main()
{
 printf("Witaj, \t ");
 printf("studencie");
 printf("\n");
 return 0;
}
```
4.
```
#include <stdio.h>

int main()
{
 int a;
 a = 2;
 printf("liczba jest rowan %d\n",  a);
 return 0;
}
```
5.
```
#include <stdio.h>

int main()
{
 int a;
 float b;
 a = 2;
 b = 4.2;
 printf("liczba %4d jest mniejsza od liczby %+8.3f \n",  a, b);
 return 0;
}
```
6.
```
#include <stdio.h>

int main()
{
 char znak;
 int a;
 float b;
 double c;
 znak = 'A';
 a = 21;
 b = 88.82;
 c = 123.456;
 printf("Wartosci zmiennych wynosza: \n");
 printf(" znak:  %c\n calkowita:  %d\n rzeczywista:  %f\n
 podwojnej precyzji:  %f\n",  znak, a, b, c);
 return 0;
 }
```
7.
```
#include <stdio.h>

int main()
{
 int x, y;
 x = 22;
 y = 7;
 printf("x =  %d\n y = %d\n",  x, y);
 printf("suma = %d\t roznica = %d\n iloczyn = %d\t iloraz = %d\n",
 x+y, x-y, x*y, x/y);
 printf("reszta z dzielenia = %d\n", x%y);
 return 0;
}
```
8.
```
#include <stdio.h>

int main()
{
 int x ;
 const y = 7;
 x = 22;
 printf("x =  %d\n y = %d\n",  x, y);
 printf("suma = %d\t roznica = %d\n iloczyn = %d\t iloraz = %d\n",
 x+y, x-y, x*y, x/y);
 printf("reszta z dzielenia = %d\n", x%y);
 return 0;
}
```
9.
```
#include <stdio.h>

int main()
{
 int a, b;
 printf("podaj liczbe: ");
 scanf("%d", &a);
 b = a*a;
 printf("kwadrat liczby  %d  wynosi:  %d\n", a, b);
return 0;
}
```
10.
```
#include <stdio.h>

int main()
{
 char znak;
 int calkowita;
 float rzeczywista;
 double podwojnej_precyzji;
 scanf("%c%d%f%lf", &znak, &calkowita, &rzeczywista,
                    &podwojnej_precyzji);
 printf("Wartosci zmiennych wynosza: \n");
 printf(" %c%8d %e\n\t%f\n", znak, calkowita, rzeczywista,
 podwojnej_precyzji );
 return 0;
}
```

Instrukcje sterujące

11.
```
#include <stdio.h>

int main()
{
 int i;
 for (i = 1; i < 10; i++)
     printf("%2d drzewo\n ", i);
 printf("wystarczy\n");
 return 0;
 }
```

12.

```
#include <stdio.h>

int main()
{
 int liczba;
 printf("podaj liczbe\n ");
 scanf("%d", &liczba);
 if(liczba == 10) printf("Brawo trafiles wdziesiatke!\n");
 if(liczba != 10) printf("Nie trafiles.\n");
 return 0;
}
```

13.

```
#include <stdio.h>

int main()
{
 double liczba;
 printf("podaj liczbe\n ");
 scanf("%lf", &liczba);
 if(liczba <= 0.0)
 printf("wartosc bezwzgledna wynosi %f\n", -liczba);
else
 printf("wartosc bezwzgledna wynosi %f\n", liczba);
return 0;
}
```

14.

```
#include <stdio.h>

int main()
{
 double element, epsilon, suma, nr_elementu;
 printf("Sumuje elementy szeregu arytmetycznego\n ");
 printf("z dokladnoscia do epsilon\n ");
 printf("podaj epsilon\n ");
 scanf("%lf", &epsilon);
 suma = 0.0;
 nr_elementu = 1.0;
do
  {
   element = 1.0/nr_elementu++;
   suma += element;
   }
   while(element > epsilon);
 printf("suma %d wyrazow szeregu z dokladnoscia do %8.4f
 wynosi %12.4e\n", (int)nr_elementu – 1, epsilon, suma);
return 0;
}
```

15.

```
#include <stdio.h>

int main()
{
 int c;
 printf("Wprowadz liczbe: \n ");
 while((c = getchar()) != EOF)
 {
  if (( c == '\n') || (c == '\r' ))
    continue;
  if (( c < '0' ) || ( c > '9'))
  {  printf("Musisz podac liczbe jednocyfrowa \n");
     printf("Wprowadz liczbe: \n");
     continue;
  }
  switch(c)
   {   case '9':
       case '8':
          printf("Liczba nie jest w przedziale 1 – 7\n");
          break;
       default:
          goto koniec;
       case '1':
          printf("Poniedzialek\n");
          break;
       case '2':
          printf("Wtorek\n");
          break;
       case '3':
          printf("Sroda\n");
          break;
       case '4':
          printf("Czwartek\n");
          break;
       case '5':
          printf("Piatek\n");
          break;
       case '6':
          printf("Sobota\n");
          break;
       case '7':
          printf("Niedziela\n");
          break;
    }
    printf("Wprowadz liczbe:  /n");
  }
  goto zakoncz;
koniec:
  printf("Zero jest bledne, konczenie programu... \n");
zakoncz:
return 0;
}
```

Zadania

1. Wypisać na ekranie wszystkie znaki ASCII wraz z ich numerami.
2. Sprawdzić czy dana liczba jest parzysta.
3. Oblicz średnia arytmetyczna liczb podanych z klawiatury.

Literatura:

1. Brian W. Kernigham, Dennis M. Ritchie, Język ANSI C, WNT, Warszawa, 2004
2. S. Oualline, Język C Programowanie, HELION(O'REILLY), Gliwice, 2003

Instytut Fizyki
Uniwersytet Jana Kochanowskiego
Kielce

Studia Informatyka
Rok I-szy. Języki i Środowisko Programisty.

ZAJĘCIA – PROGRAMOWANIE C – OPERACJE NA PLIKACH.

Przykład 1

```
#include <stdio.h>
/*zawartosc zbioru czytaj.dat wyglada
następująco:
l 2 3 4
5 6 7 8
*/
int main( )
{
int a, b, c, d, e, f, g, h;
FILE *czytany_plik;
czytany_plik = fopen("F:\\czytaj.dat", "rt");
fscanf(czytany_plik, "%d%d%d%d", &a, &b,
&c, &d);
fscanf(czytany_plik, "%d%d%d%d", &e, &f, &g,
&h);
printf("%d %d %d %d %d %d %d %d\n", a, b,
c, d, e, f, g, h);
rewind(czytany_plik);
fscanf( czytany_plik, "%d%d%d%d", &a, &b,
&c, &d);
printf("%d %d %d %d\n", a, b, c, d);
fclose( czytany_plik);
return 0;
}
```

Przykład 2

```
#include <stdio.h>
#include <io.h>
int czytaj_wektor(double *A, int
maksymalny_rozmiar)
{
int i;
FILE *plik_dane;
plik_dane = fopen("e:\\czytaj.mat", "rt");
i = 0;
while(!feof(plik_dane) &&
i<maksymalny_rozmiar)
{
i++;
fscanf(plik_dane, "%lf", &A[i]);
}
fclose(plik_dane);
return i; }
/* teraz program */
#define max_rozmiar 100
int main( )
{
double A[max_rozmiar];
int i, rozmiar;
rozmiar = czytaj_wektor(A, max_rozmiar);
for(i = 0; i<rozmiar; i++)
printf("%12.6f\n",A[i]);
return 0;
}
```

Przykład 3

```
#include <stdio.h>
/* program ilustruje dopisywanie do
istniejącego zbioru*/
void main(void)
{
FILE *zbior;
zbior = fopen(" F:\\dopisz", "a+");
fprintf(zbior, "to zostało dopisane\n");
fclose(zbior);
}
```

Przykład 4

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main(int argn, char *argv[])
{
float a, b, c, d;
char nazwa_pliku[50];
FILE *pomiary1, *pomiary2;
pomiary1 = fopen(argv[1], "wt");
printf("podaj nazwę pliku: ");
scanf("%s", nazwa_pliku);
pomiary2 = fopen(nazwa_pliku, "wt");
a= 1.0;
b = 2.5;
c = 3.14;
d = 4567.1;
fprintf(pomiary1, "%f %f %f %f", a, b, c, d);
fprintf(pomiary2, "%f %f %f %f", a, b, c, d);
fclose(pomiary1);
fclose(pomiary2);
return 0;
}
```

Przykład 5

```
/*Otwieranie i zamykanie pliku ze
sprawdzeniem jego istnienia*/
#include <stdio.h>
enum{SUKCES, FIASKO};
/*SUKCES=0,FIASKO=1*/
int main(void)
{
FILE *fptr;
char filename[ ] = "f:\\plik.txt";
int zwrot_z_main = SUKCES;
if ((fptr = fopen(filename, "r")) == NULL)
{
printf("Nie moge otworzyc plik: %s.\n",
filename);
zwrot_z_main = FIASKO;
} else {
printf("Wartosc wskaznika fptr: 0x%p\n", fptr);
if (fclose(fptr) == 0) printf("Plik zamkniety.");
} return(zwrot_z_main);
}
```

Przykład 6

```
/*Odczyt i zapis z-do pliku znak-po-znaku*/
#include <stdio.h>
enum{SUKCES, FIASKO};
void CharReadWrite(FILE *fin, FILE *fout);
main(void)
{
FILE *fptr1, *fptr2;
char filename1[ ] = "f:\\plik_wy.txt";
char filename2[ ] = "f:\\plik_we.txt";
int reval = SUKCES;
if((fptr1=fopen(filename1, "w")) == 0)
{ printf("Nie moge otworzyc pliku %s.\n",
filename1);
reval = FIASKO;
} else if((fptr2 = fopen(filename2, "r")) == 0)
 {printf("Nie moge otworzyc pliku %s.\n",
filename2);
reval = FIASKO;
} else {
CharReadWrite(fptr2, fptr1);
fclose(fptr1);
fclose(fptr2);
}
return reval;
}
/* def. funkcji*/
void CharReadWrite(FILE *plik_we, FILE
*plik_wy)
{
int c;
while ((c = fgetc(plik_we)) != EOF)
{
putchar(c);
fputc(c, plik_wy);
}
}
```

Przykład 7

```
/* Wczytywanie i zapis wiersz po wierszu. */
#include <stdio.h>
enum {SUCCESS, FAIL, MAX_LEN = 81}; /*
maks. długość 81 znaków */
void LineReadWrite(FILE *fin, FILE *fout);
main(void)
{
FILE *fptr1, *fptr2;
char filename1[ ] = "f://plik_we.txt";
char filename2[ ]= "f://plik_wy.txt";
int reval = SUCCESS;
if ((fptr1 = fopen(filename1, "w")) == NULL){
printf("Cannot open %s for writing.\n",
filenamel); /* Błąd otwarcia pl. Wyjściowego */
reval = FAIL;
} else if ((fptr2 = fopen(filename2, "r")) ==
NULL){
printf("Cannot open %s for reading.\n",
filename2); /* Błąd otwarcia pliku
Wejściowego */
reval = FAIL;
}else{
LineReadWrite(fptr2, fptr1);
fclose(fptr1);
fclose(fptr2);
}
return reval;
}        /* definicja funkcji */
void LineReadWrite(FILE *fin, FILE *fout)
{     char buff[MAX_LEN];     /* bufor na 81
znaków */
/* dopóki funkcja fgets() nie zwróci wskaźnika
pustego */
while (fgets(buff, MAX_LEN, fin) != NULL) {
fputs(buff, fout);
printf("%s", buff);
}
}
```

Przykład 8

```
/* Wczytywanie I zapis, blok po bloku */
#include <stdio.h>
enum {SUCCESS, FAIL, MAX_LEN = 80};
void BlockReadWrite(FILE *fin, FILE *fout);
int ErrorMsg(char *str);
int main( )
{ FILE *fptr1, *fptr2;
char filename1[]= "e:\\outhaiku.txt";
char filename2[]= "e:\\haiku.txt";
int reval = SUCCESS;
if ((fptr1 = fopen(filename1, "w")) == NULL){
reval = ErrorMsg(filename1);
} else if ((fptr2 = fopen(filename2, "r")) ==
NULL){
reval = ErrorMsg(filename2);
} else {
  BlockReadWrite(fptr2, fptr1);
  fclose(fptrl);
fclose(fptr2);
}
return reval;
}
/* function definition */
void BlockReadWrite(FILE *fin, FILE *fout)
{ int num;
 char buff[MAX_LEN + 1];
   while (!feof(fin)){
   num = fread(buff, sizeof(char), MAX_LEN,
 fin);
   buff[num * sizeof(char)] = '\0';
   printf("%s", buff);
   fwrite(buff, sizeof(char), num, fout);
                              } }
/* function definition */
int ErrorMsg(char  *str)
{
  printf("Cannot open %s.\n", str);
  return FAIL;
}
```

Instytut Fizyki
Uniwersytet Jana Kochanowskiego
Kielce

Studia Informatyka
Rok I-szy. Języki i Środowisko Programisty.

ZAJĘCIA --TeX – WPROWADZENIE.

TeX jest systemem składu tekstu za pomocą którego otrzymujemy dokumenty najwyższej jakości. LaTeX natomiast jest zbiorem instrukcji korzystających z TeX-a, mających maksymalnie autorom ułatwić składanie własnych dokumentów.

Ćwiczenie 1: W edytorze ASCII stwórz plik tekst1.tex

```
% Preambuła
\documentclass[a4paper,11pt]{article}

\usepackage[polish]{babel}
\usepackage[cp1250]{inputenc}
\usepackage[T1]{fontenc}
\usepackage{setspace}
\usepackage[a4paper,left=2.5cm,right=2.5cm,top=2c
m,bottom=2cm]{geometry}
%pakiet do manipulowania rysunkami i tabelami
\usepackage{epsfig}

% Część główna
\begin{document}


\title{Wprowadzenie do \LaTeX-a}
\author{Jan Kowalski}
\date{\today}
\maketitle

Treść.

 To jest pierwszy paragraf. Nie  ma znaczenia
liczba
odstępów  między wyrazami. Dobrym nawykiem
jest umieszczanie
zdań w osobnych liniach.

Aby zacząć następny (drugi już) paragraf wystarczy
dodać co
najmniej jedną pustą linię.

Początkowe zdanie.

\section{Wstęp}

Treść wstępu.

\section{Sekcja główna}

Treść sekcji głównej i mniejsze sekcje wchodzące w
jej skład.

\subsection{Podpunkt pierwszy}

Treść pierwszego podpunktu.

\subsubsection{Podpodpunkt}

Treść podpodpunktu.

\subsection{Podpunkt drugi}

Treść drugiego podpunktu.

To \emph{słowo} jest wyróżnione
(tak właśnie \LaTeX{} rozumie wyróżnianie słów).
\texttt{To zdanie jest napisane czcionką
maszynową.}
Z kolei to {\sf słowo} i to \textsf{słowo} jest
napisane
czcionką bezszeryfową.

{\Large Ten akapit jest trochę większy.
Możemy nadal stosować inne czcionki, np. {\bf
pogrubioną}
i będą one również powiększone.}

W tym akapicie {\small niektóre słowa} są mniejsze.
Do tego {\small mogą być napisane {\sc kapitalikami
lub być pisane {\it kursywą}}}.


\end{document}
```

Ćwiczenie 2.
Następnie przejdź na stronę UW
http://wazniak.mimuw.edu.pl/index.php?title=%C5%9Arodowisko_programisty/Sk%C5%82adanie_dokument%C3%B3w_-_Latex
i wykonaj znajdujące się tam polecenia kompilacji (**latex, pdflatex**) oraz polecenia **xdiv, divps, gv** w systemie Windows i Linux.

Ćwiczenie 3.
Zapoznaj się dokładnie z przykładami umieszczonymi na powyższej stronie a następnie je wszystkie skompiluj w jednym dokumencie.

Wprowadzenie do LaTeX-a

Jan Kowalski

30 listopada 2011

Treść.

To jest pierwszy paragraf. Nie ma znaczenia liczba odstępów między wyrazami. Dobrym nawykiem jest umieszczanie zdań w osobnych liniach.

Aby zacząć następny (drugi już) paragraf wystarczy dodać co najmniej jedną pustą linię.

Początkowe zdanie.

1 Wstęp

Treść wstępu.

2 Sekcja główna

Treść sekcji głównej i mniejsze sekcje wchodzące w jej skład.

2.1 Podpunkt pierwszy

Treść pierwszego podpunktu.

2.1.1 Podpodpunkt

Treść podpodpunktu.

2.2 Podpunkt drugi

Treść drugiego podpunktu.

To *słowo* jest wyróżnione (tak właśnie LaTeX rozumie wyróżnianie słów). `To zdanie jest napisane czcionką maszynową.` Z kolei to słowo i to słowo jest napisane czcionką bezszeryfową.

Ten akapit jest trochę większy. Możemy nadal stosować inne czcionki, np. **pogrubioną** i będą one również powiększone.

W tym akapicie niektóre słowa są mniejsze. Do tego mogą być napisane KAPITALIKAMI LUB BYĆ PISANE *kursywą*.