Instrukcja do zajęć laboratoryjnych

Język ANSI C (w systemie LINUX)

wersja: 1.24

Nr ćwiczenia:	4	
Temat:	**Sterowanie procesem budowania programów – GNU make**	
Cel ćwiczenia:	Celem ćwiczenia jest zapoznanie się z zasadą działania programu `make`, służącego do sterowania procesem budowania programów.	
Wymagane przygotowanie teoretyczne:	Informacje podane na wykładzie. Podstawy pracy z systemem LINUX.	
Sposób zaliczenia:	Sprawozdanie w formie pisemnej.	[ ]
	Pozytywna ocena ćwiczenia przez prowadzącego pod koniec zajęć.	[X]

1. Konwencje przyjęte w instrukcji

`Czcionka o stałej szerokości`

Nazwy programów, poleceń, katalogów, wyniki działania wydawanych poleceń.

`Czcionka o stałej szerokości pogrubiona`

Podaje tekst, który należy dosłownie przepisać. W przypadku plików źródłowych wyróżnia istotniejsze fragmenty.

`Czcionka o stałej szerokości kursywą`

Tekst komentarza w przykładowych sesjach przy terminalu.

`Czcionka o stałej szerokości kursywą pogrubiona`

Wyróżnia istotniejsze fragmenty wyników działania wydawanych poleceń.

2. Przykładowe programy

Wszystkie wykonywane ćwiczenia będą oparte o następujące, podane niżej programy. Programu `hello.c` nie trzeba chyba nikomu przedstawiać. Programy `calc.c`, `mymath.c`, `mymath.h` skompilowane wspólnie utworzą jeden wynikowy program wykonywalny obliczający sumę, różnicę, iloczyn oraz iloraz dwóch liczb.

Należy zwrócić uwagę na dyrektywy `#ifndef`, `#define` oraz `#endif` w pliku `mymath.h`. Można powiedzieć, że „owijają" one właściwe deklaracje w pliku nagłówkowym. Używanie ich jest często stosowana praktyką, gdyż pozwala uniknąć wielokrotnego dołączania tego samego pliku nagłówkowego w programie składającym się z wielu plików źródłowych.

```
/*
hello.c
*/

#include<stdio.h>
int main (void)
{
  printf("\n%s\n\n", "Hello, world!");
  return 0;
}
```

```
/*
mymath.h
*/

#ifndef _MYMATH
#define _MYMATH

double myadd (double, double);
double mysub (double, double);
double mymul (double, double);
double mydiv (double, double);

#endif  /* _MYMATH */
```

```
/*
mymath.c
*/

#include "mymath.h"

double myadd (double a, double b) {
  return a + b;
}

double mysub (double a, double b) {
  return a - b;
}

double mymul (double a, double b) {
  return a * b;
}

double mydiv (double a, double b) {
  return a / b;
}
```

```
/*
calc.c
*/

#include <stdio.h>
#include "mymath.h"
```

```
int main (void)
{
  double a=5, b=20;
  printf("\na = %8.2f\nb = %8.2f\n\n",a,b);
  printf ("a + b = %8.2f\n", myadd(a,b));
  printf ("a - b = %8.2f\n", mysub(a,b));
  printf ("a * b = %8.2f\n", mymul(a,b));
  printf ("a / b = %8.2f\n", mydiv(a,b));
  printf("\n");
  return 0;
}
```

3. Zasada działania oraz struktura plików dla programu make

Program `make` jest programem, który w zdecydowany sposób ułatwia pisanie złożonych programów w różnych językach (w systemie UNIX/LINUX najczęściej będą to języki `C/C++`). Jeśli tworzysz aplikację składającą się z wielu plików źródłowych ich „ręczna" kompilacja i konsolidacja może być dosyć uciążliwa. Narzędzie `make` ułatwia znacznie pracę programisty.

Program ten korzysta ze specjalnego pliku tekstowego (tzw. *makefile*), w którym programista zapisuje w ustandaryzowany sposób co i w jakiej kolejności należy wykonać, aby zbudować dany projekt. Kompilacja całego (nawet bardzo złożonego) projektu wymaga wówczas wydania tylko jednego lub maksymalnie kilku poleceń z konsoli a całą resztę zrobi za nas program `make`. Oczywiście warunkiem, aby budowanie projektu było aż do takiego stopnia procesem zautomatyzowanym jest utworzenie, a później w miarę potrzeb uaktualnianie, wspomnianego już pliku *makefile*.

Należy również podkreślić, że możliwe jest przygotowanie jednego wspólnego pliku *makefile*, z pomocą którego będzie można budować więcej niż jeden projekt, przy czym projekty te wcale nie muszą być w jakikolwiek sposób ze sobą powiązane. Oczywiście w praktyce programiści raczej do każdego projektu przygotowują niezależny plik *makefile*. Wspominamy tutaj o takiej możliwości tylko dlatego, że jeden z przykładowych (zamieszczonych w dalszej części instrukcji) plików *makefile* kompiluje trzy przykładowe, bardzo proste programy (`hello, calc, calclib`), które działają całkowicie niezależnie.

Program `make`, gdy zostanie wywołany bez żadnych argumentów, poszukuje na dysku plików `GNUmakefile, makefile` lub `Makefile` (w tej właśnie kolejności). Jeżeli plik ma inną nazwę, to wywołanie programu `make` powinno odbyć się w następujący sposób:

```
make –f nasz_plik_makefile
```

Inne najczęściej używane opcje programu `make` to:

- `–n` tylko wyświetla polecenia, ale ich nie wykonuje. Użyteczne podczas testowania pliku *makefile*
- `–s` program `make` nie wypisuje wykonywanych poleceń
- `–w` wyświetlane są nazwy katalogów podczas ich zmieniania przez program `make`
- `–Wplik` wykonuje `make`, tak jakby `plik` był zmodyfikowany. Opcja bardzo użyteczna podczas testowania pliku *makefile*
- `–i` ignoruje wszystkie błędy (normalnie po napotkaniu pierwszego błędu przetwarzanie jest zatrzymywane)

W pliku *makefile* występują tzw. reguły (ang. *rule*) mające następującą, jednolitą formę (zapis `<znak TAB>` oznacza, że w tym miejscu stoi znak tabulatora):

```
cel: zależność [zależność] [ ...]
<znak TAB> polecenie
<znak TAB> polecenie
<znak TAB> [...]
```

Uwaga: pierwszym znakiem w poleceniu **MUSI** być znak tabulatora. Nie może to być np. 8 spacji. Jeżeli omyłkowo zamiast znaku tabulatora wpiszemy spacje, program `make` wyświetli komunikat `missing separator` i zakończy działanie.

Znaczenie poszczególnych pozycji w pliku *makefile* jest następujące:

- `cel` – (ang. *target*) jest zazwyczaj binarnym lub obiektowym plikiem, który chcemy utworzyć,
- `zależność` – (ang. *prerequisite*) do utworzenia *celu* wymagane są zwykle pliki źródłowe. Nazywamy je *zależnościami*. Gdy plików tych jest kilka, to są one oddzielone spacjami. W ogólności w danym *celu* może być wiele *zależności* lub też może ich nie być wcale. Gdy ich nie ma, to wówczas mówimy o tzw. sztucznych celach (patrz opis niżej),
- `polecenia` – (ang. *command*) są krokami (np. wywołania kompilatora lub polecenia powłoki), które należy wykonać, aby utworzyć *cel*.

Reguła „mówi" programowi `make` dwie rzeczy: kiedy *cel* jest nieaktualny i jak uaktualnić *cel*. Kryterium według jakiego `make` uznaje *cel* wynikowy za nieaktualny jest określony w liście *zależności*, która składa się z nazw plików oddzielonych spacjami.

Cel jest przestarzały (ang. *obsolete*) jeżeli nie istnieje lub jeśli jest starszy od któregokolwiek z plików wymienionych w liście *zależności*. Program `make` porównuje daty ostatniej modyfikacji plików. Idea narzędzia `make` jest taka: zawartość pliku określonego jako *cel* zależy od zawartości plików określonych w liście *zależności*, więc jeśli którykolwiek z tych plików się zmieni, zawartość pliku stanowiącego *cel* powinna również się zmienić. Sposób w jaki należy uaktualnić obiekt wynikowy jest określony przez *polecenie* (jedno lub kilka).

Jeżeli lista *zależności* jest pusta, mówimy wówczas o tzw. sztucznych celach (ang. *phony targets*) Zostały one tak nazwane, gdyż nie odpowiadają prawdziwym plikom dyskowym. Sztuczne cele, podobnie jak normalne, zawierają polecenia, które program `make` powinien wykonać. W poniższym pliku takim sztucznym celem jest cel o nazwie `kasuj`.

Pierwszy cel zdefiniowany w pliku *makefile* jest celem domyślnym. Gdy w wywołaniu programu `make` nie podamy jawnie nazwy celu, to on właśnie zostanie wykonany jako pierwszy.

Poniższy przykład jest plikiem programu `make`, który kompiluje program `hello.c`. Znaki `#` w plikach *makefile* są znakami początku komentarza i wszystko, co po nic występuje jest oczywiście pomijane przy przetwarzaniu.

```
#
# Służy do kompilowania pliku źródłowego hello.c
#

# Celem wynikowym jest plik hello. Jest on zależny od pliku hello.c. Aby
# zbudować cel hello należy wykonać dwa polecenia (wywołania kompilatora
# gcc).
# Plik hello będzie uważany za nieaktualny, gdy nie istnieje jeszcze lub
# też plik hello.c ma datę modyfikacji późniejszą niż plik hello.
hello: hello.c
```

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

# Sztuczny cel.
kasuj:
   rm hello.o
```

Poniżej pokazano w jaki sposób odbywa się praca z tym programem:

```
Uruchamiamy program make.
$ make
gcc -c hello.c -o hello.o
gcc hello.o -o hello

Program make jest na tyle „inteligentny", że powtórne jego wywołanie nie
kompiluje programu od nowa. Program make potrafi mianowicie określić,
które pliki były zmienione i ponownie je kompilować tylko wówczas, gdy to
jest wymagane.
$ make
make: `hello' is up to date.

Za pomocą polecenia touch modyfikujemy datę ostatniej modyfikacji pliku
hello.c. Tym razem program make ponownie go kompiluje.
$ touch hello.c
$ make
gcc -c hello.c -o hello.o
gcc hello.o -o hello

Aby wywołać regułę niejawną trzeba ją podać jako parametr wywołania
programu make. Reguła ta zostanie wykonana za każdym razem ponieważ plik
kasuj nigdy nie zostanie utworzony.
$ make kasuj
rm hello.o

W wywołaniu programu make można podać nazwę celu (tu: hello). Nie jest to
jednak w tym konkretnym przypadku konieczne, gdyż cel hello jest celem
domyślnym i zostanie zbudowany nawet wówczas, gdy nie wyspecyfikujemy go
jawnie w wywołaniu programu make.
$ touch hello.c
$ make hello
gcc -c hello.c -o hello.o
gcc hello.o -o hello
```

Rozważmy inny przykład. Zdefiniowano w nim trzy cele niejawne:

```
#
# Makefile z trzema niejawnymi celami.
#

komp:
   @echo "Wykonuję kompilację ..."

inst:
   @echo "Wykonuję instalację ..."

inst_dok:
   @echo "Wykonuję instalację dokumentacji ..."
```

Wydajmy teraz polecenie `make` - raz bez parametrów a raz z podanymi parametrami wywołania:

```
Polecenie make wydajemy bez parametrów. Domyślnie wykonywany jest cel
pierwszy.
$ make
Wykonuję kompilację ...
```

```
Teraz z kolei jawnie określamy cele. Cele zostały wykonane w określonej
przez nas kolejności.
$ make inst_dok komp inst
Wykonuję instalację dokumentacji ...
Wykonuję kompilację ...
Wykonuję instalację ...
```

O sile programu `make` świadczy umiejętność honorowania zależności między celami. Ma to zasadnicze znaczenie w złożonych projektach, gdzie kompilacja plików źródłowych musi odbywać się w ściśle określonej kolejności. Rozważmy więc kolejny przykład pliku *makefile*, gdzie tym razem między poszczególnymi celami określono pewne zależności. Mamy w nim zapisane, że instalacja dokumentacji musi odbywać się po instalacji systemu, a ta wymaga wcześniejszej kompilacji projektu. Plik *makefile* wyglada teraz następująco:

```
#
# Demonstracja umiejętności honorowania zależności między celami
#

komp:
   @echo "Wykonuję kompilację ..."

inst: komp
   @echo "Wykonuję instalację ..."

inst_dok: inst
   @echo "Wykonuję instalację dokumentacji ..."
```

Praca z tym plikiem może teraz wyglądać następująco:

```
Polecenie make wydajemy z parametrem inst_dok. Program make umie
rozstrzygnąć w jakiej kolejności oraz które cele mają zostać zbudowane.
Zamin więc zostanie zbudowany cel inst_dok, wcześniej zostaną zbudowane
cele komp oraz inst.
$ make inst_dok
Wykonuję kompilację ...
Wykonuję instalację ...
Wykonuję instalację dokumentacji ...
```

4. Bardziej złożone przykłady pracy z programem make

4.1. Typowe sztuczne cele

Poniżej pokazano przykładowy plik *makefile* programu `make`, który zawiera dość powszechnie używane w praktyce nazwy sztucznych celów.

```
#
# Makefile - zwyczajowo używane sztuczne cele.
#

default:
   @echo "Opcja wybierana domyślnie."

install:
   @echo "Instalacja systemu."

all:
   @echo "Budowa całego projektu."

clean:
```

```
   @echo "Usuwanie plików obiektowych (zwykle *.o)."

doc:
   @echo "Instalacja dokumentacji."
```

4.2. Przykład – kompilacja programu składającego się z wielu plików źródłowych

Poniżej pokazano przykładowy plik *makefile* programu `make`, który z trzech plików źródłowych tworzy plik wykonywalny o nazwie `calc`:

```
#
# Makefile
#

buduj:  calc.c mymath.c mymath.h
   gcc mymath.c calc.c -o calc

czysc:
   rm -f calc
```

4.3. Przykład – kompilacja prgramu `hello.c` (ale nieco inaczej niż poprzednio)

Poniżej pokazano przykładowy plik *makefile* programu `make`, który tworzy plik wykonywalny `hello`. W stosunku do pliku *makefile* zamieszczonego w poprzednim rozdziale dodano kilka celów oraz zmieniono sposób budowania celu `hello`. Mamy tutaj trzy cele (`install`, `hello`, oraz `hello.o`), które wywoływane są przez program `make` automatycznie jeden po drugim. Odbywa się to następująco:

- celem domyślnym jest cel o nazwie `install`,
- ponieważ plik `hello` nie istnieje program `make` poszukuje w pliku *makefile* celu o takiej nazwie (i odnajduje go),
- aby zbudować cel o nazwie `hello` potrzebny jest plik `hello.o`. W tym momencie nie ma go jeszcze na dysku. Program `make` szuka więc celu o nazwie `hello.o` (i odnajduje go) ,
- aby zbudować cel o nazwie `hello.o` potrzebny jest plik `hello.c`. Plik ten istnieje na dysku. Wykonywane jest więc polecenie dla tego celu. Na dysku powstaje plik `hello.o`,
- program `make` „wraca” do nieukończonego jeszcze celu `hello`. Ponieważ tym razem na dysku istnieje już pliki `hello.o` budowany jest cel o nazwie `hello` (na dysku powstaje plik wykonywalny `hello`) ,
- program `make` „wraca” do nieukończonego jeszcze celu `install`. Ponieważ tym razem na dysku istnieje już plik `hello`, wykonywane są polecenia w tym celu (tu: wypisanie tekstu na ekranie za pomocą polecenia `echo`).

W pliku *makefile* zdefiniowano jeszcze dwa cele: `dist` oraz `uninstall`. Ich przeznaczenia oraz działania można się łatwo domyślić:

```
#
# Makefile
#

install: hello
   @echo "Instalacja programu: hello"
```

```
hello: hello.o
   gcc hello.c -o hello

hello.o: hello.c
   gcc -c hello.c -o hello.o

dist:
   @echo "Tworzenie dystrybucji:"
   $(RM) hello *.o
   tar czvf hello.tar.gz hello.c Makefile

uninstall:
   @echo "Czyszczenie:"
   rm hello hello.o hello.tar.gz
```

Na ekranie komputera powinniśmy zaobserwować obraz podobny do poniższego:

```
Zaczynamy. Na dysku mamy tylko plik makefile oraz plik źródłowy hello.c
$ ls -l
total 12
-rw-r-----    1 artur    students      303 Oct 11 07:47 Makefile
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello.c

Zaczynamy więc budowę.
$ make
gcc -c hello.c -o hello.o
gcc hello.c -o hello
Instalacja programu: hello

Próbujemy ponownie zbudować cel. Ponieważ jednak program wykonywalny hello
oraz plik obiektowy hello.o są aktualne, wiec nie ma potrzeby ich
przebudowywać.
$ make
Instalacja programu: hello

Na dysku pojawiły się pliki hello.o oraz hello.
$ ls -l
-rw-r-----    1 artur    students      303 Oct 11 07:47 Makefile
-rwxr-xr-x    1 artur    students     4819 Oct 11 16:34 hello
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello.c
-rw-r--r--    1 artur    students      936 Oct 11 16:34 hello.o

Wykonujemy cel uninstall. Z dysku kasowane są pliki hello, hello.o oraz
hello.tar.gz
$ make uninstall
Czyszczenie:
rm -f hello hello.o hello.tar.gz

Wykonujemy cel dist. Z dysku kasowane jest plik hello.o oraz tworzone jest
archiwum o nazwie hello.tar.gz
$ make dist
Tworzenie dystrybucji:
rm -f hello *.o
tar czvf hello.tar.gz hello.c Makefile
hello.c
Makefile

Na dysku pozostaje tylko plik źródłowy, plik makefile oraz archiwym.
$ ls -l
-rw-r-----    1 artur    students      303 Oct 11 07:47 Makefile
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello.c
-rw-r--r--    1 artur    students      395 Oct 11 16:26 hello.tar.gz
```

4.4. Bardziej rozbudowany przykład. Zmienne automatyczne i predefiniowane

Poniżej pokazano przykładowy plik *makefile* programu `make`. Jest on już nieco bardziej rozbudowany niż te pokazane wcześniej. Pojawiły się tutaj tzw. *zmienne automatyczne*, których wartości są obliczane za każdym razem, gdy reguła jest wykonywana. Dodatkowo użyto również tzw. *zmiennych predefiniowanych*, które są używane albo jako nazwy programów, albo do przekazywania flag i argumentów do tych programów.

Poniżej pokazano kilka najczęściej używanych zmiennych automatycznych oraz zmiennych predefiniowanych. Ich kompletną listę można oczywiście znaleźć w dokumentacji.

Zmienne automatyczne:

- `$@` – reprezentuj nazwę pliku celu w regule,
- `$*` – reprezentuje nazwę podstawową (rdzeń) celu. Powinien być używany raczej tylko w regułach niejawnych. Więcej szczegółów znajdziesz w dokumentacji (`info make`),
- `$<` – reprezentuje nazwę pliku pierwszej zależności,
- `$^` – oddzielona spacjami lista wszystkich zależności,
- `$?` – j.w. ale tylko te zależności, które są nowsze niż cel.

Predefiniowane zmienne nazw programów i flag:

- `AR` – nazwa programu do tworzenia archiwum (domyślnie: `ar`),
- `CC` – nazwa kompilatora C (domyślnie: `cc`),
- `RM` – nazwa programu do usuwania plików (domyślnie: `rm -f`),
- `CFLAGS` – flagi dla kompilatora C (domyślnie: brak wartości).

```
#
# Kompiluje 3 przykładowe proste programiki (nie są one ze sobą powiązane)
#   1. klasyczny "Hello, world"
#   2. program wieloplikowy (calc.c, mymath.c, mymath.h)
#   3. jak pkt. 2 ale łączone z biblioteka statyczna libmymath.a

# Zmienne predefiniowane
CFLAGS  = -g
CC      = cc

# Zmienna zdefiniowana przez użytkownika (nazwa może być dowolna).
PROGS   = hello calc calclib

# Reguła domyślna (bo jest w pliku jako pierwsza).
all: $(PROGS)

# Reguła niejawna.
# Mówi ona, że dla każdej nazwy pliku z rozszerzeniem .c
# należy utworzyć plik obiektowy z rozszerzeniem .o
.c.o:
   $(CC) $(CFLAGS) -c $*.c

# Za $^ zostanie podstawiona lista zależności, czyli 'calc.c mymath.c'
# a za $@ zostanie podstawiona nazwa celu, czyli 'calc'.
calc: calc.c mymath.c
   $(CC) $(CFLAGS) $^ -o $@

# Za $< zostanie podstawiona nazwa pierwszej zależności, czyli 'calc.c'
```

```
# Za $@ zostanie podstawiona nazwa celu, czyli 'calclib'.
calclib: calc.c libmymath.a
   $(CC) $(CFLAGS) $< -o $@ -I. -L. -lmymath

# Za $< zostanie podstawiona nazwa pierwszej zależności (tutaj jest
# zresztą tylko jedna), czyli 'mymath.c'.
libmymath.a: mymath.c
   $(CC) $(CFLAGS) -c $< -o libmymath.o
   $(AR) rsc libmymath.a libmymath.o

# Pliki „z falką” na końcu nazwy tworzone są często przez edytory
# tekstów, które w ten sposób zapisują poprzednią wersję pliku (przed
# zapisaniem bieżących zmian).
clean:
   $(RM) $(PROGS) *.o *.a *~ *.out

# Plik hello.c skompiluje się poprawnie, gdyż zadziałają
# tutaj dwie reguły niejawne.
# Innymi słowy: są zapisane zależności, ale nie ma reguł
# budowania celów.
#
# 1-sza reguła niejawna:
#   "Dla każdego pliku obiektowego plik.o należy poszukać pliku
#    źródłowego plik.c i zbudować plik obiektowy za pomocą polecenia:
#    cc -c plik.c -o plik.o"
#
# 2-ga reguła niejawna:
#   "Każdy plik obiektowy plik.o jest scalany w ostateczny plik
#    wykonywalny za pomocą polecenia:
#    cc plik.o -o plik

hello: hello.o

hello.o: hello.c

# Można też wpisać po prostu
# hello: hello.c
```

Na ekranie komputera powinniśmy zaobserwować obraz podobny do poniższego:

```
Zaczynamy. Na dysku mamy tylko plik makefile oraz pliki źródłowe.
$ ls -l
-rw-r-----    1 artur    students     2368 Oct 11 07:47 Makefile
-rw-r-----    1 artur    students      320 Oct 11 07:47 calc.c
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello.c
-rw-r-----    1 artur    students      239 Oct 11 07:47 mymath.c
-rw-r-----    1 artur    students      180 Oct 11 07:47 mymath.h

Wydajemy polecenie make. Cały projekt zostaje zbudowany.
$ make
cc -g -c hello.c
cc   hello.o   -o hello
cc -g calc.c mymath.c -o calc
cc -g -c mymath.c -o libmymath.o
ar rsc libmymath.a libmymath.o
cc -g calc.c -o calclib -I. -L. -lmymath

Próbujemy ponownie zbudować projekt Ponieważ jednak pliki potrzebne do
zbudowania celów są aktualne, wiec nie ma potrzeby ich przebudowywać.
$ make
make: Nothing to be done for `all'.
```

Zmieniamy czas i datę aktualizacji pliku hello.c i ponownie budujmy projekt. Program make kompiluje tylko te elementy, które się zmieniły od ostatniego budowania projektu.

```
$ touch hello.c
$ make              //(lub prościej make -Whello.c)
cc -g -c hello.c
cc   hello.o   -o hello
```

Tym razem poleceniem touch zmieniamy „timestamps" pliku libmymath.a. Program make i tym razem potrafi sobie z tym poradzić. Przebudowuje tylko zmodyfikowaną bibliotekę.

```
$ touch libmymath.a
$ make
cc -g calc.c -o calclib -I. -L. -lmymath
```

*Sprawdzamy jak wygląda zawartość dysku. Pojawiły się na nim pliki obiektowe *.o, plik z biblioteką libmymath.a oraz pliki wykonywalne (hello oraz calc).*

```
$ ls -l
-rw-r-----    1 artur    students     2368 Oct 11 07:47 Makefile
-rwxr-xr-x    1 artur    students    15575 Oct 11 16:56 calc
-rw-r-----    1 artur    students      320 Oct 11 07:47 calc.c
-rwxr-xr-x    1 artur    students    15575 Oct 11 16:56 calclib
-rwxr-xr-x    1 artur    students    14075 Oct 11 16:56 hello
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello.c
-rw-r--r--    1 artur    students    10584 Oct 11 16:56 hello.o
-rw-r--r--    1 artur    students     2952 Oct 11 16:56 libmymath.a
-rw-r--r--    1 artur    students     2780 Oct 11 16:56 libmymath.o
-rw-r-----    1 artur    students      239 Oct 11 07:47 mymath.c
-rw-r-----    1 artur    students      180 Oct 11 07:47 mymath.h
```

Zmieniamy kod źródłowy biblioteki (tak na prawdę to tylko symulujemy taką zmianę poprzez opcję -Wmymath.c). Biblioteka zostanie przebudowana i dodatkowo kalkulator, który z niej korzysta również zostanie uaktualniony. Dzięki opcji -n na ekranie pojawią się tylko stosowne polecenia, ale w rzeczywistości nic się nie wykona.

```
$ make -Wmymath.c -n calclib
cc -g -c mymath.c -o libmymath.o
ar rsc libmymath.a libmymath.o
cc -g calc.c -o calclib -I. -L. -lmymath
```

Dla wersji bez jawnego wskazania celu (calclib), przybędzie do wykonania jedna czynność. Nic dziwnego: wersja „niebibliotekowa" kalkulatora korzysta również z pliku mymath.c

```
$ make -Wmymath.c -n
cc -g calc.c mymath.c -o calc
cc -g -c mymath.c -o libmymath.o
ar rsc libmymath.a libmymath.o
cc -g calc.c -o calclib -I. -L. -lmymath
```

I na koniec symulacja działania całości.

```
$ make clean
rm -f hello calc calclib *.o *.a *~ *.out

$ make -n
cc -g -c hello.c
cc   hello.o   -o hello
cc -g calc.c mymath.c -o calc
cc -g -c mymath.c -o libmymath.o
ar rsc libmymath.a libmymath.o
cc -g calc.c -o calclib -I. -L. -lmymath
```

```
Gdy jesteśmy wreszcie pewni, że program make wykona to co rzeczywiście
chcemy, uruchamiamy go. Wynik na ekranie jest analogiczny jak poprzednio.
Teraz jednak wszystkie czynności wykonują się na prawdę.
$ make
cc -g -c hello.c
cc   hello.o   -o hello
cc -g calc.c mymath.c -o calc
cc -g -c mymath.c -o libmymath.o
ar rsc libmymath.a libmymath.o
cc -g calc.c -o calclib -I. -L. -lmymath

Usuwamy wszystko oprócz plików źródłowych i pliku makefile.
$ make clean
rm -f hello calc calclib *.o *.a *~ *.out

I na koniec sprawdzamy co mamy na dysku.
$ ls -l
-rw-r-----    1 artur    students     2368 Oct 11 07:47 Makefile
-rw-r-----    1 artur    students      320 Oct 11 07:47 calc.c
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello.c
-rw-r-----    1 artur    students      239 Oct 11 07:47 mymath.c
-rw-r-----    1 artur    students      180 Oct 11 07:47 mymath.h
```

4.5. Pewna ciekawa cecha programu make

Program `make` potrafi również rozpoznać (na podstawie rozszerzenia) z jakim plikiem ma do czynienia i podjąć odpowiednią akcję. Wie na przykład, że gdy napotka plik z rozszerzeniem `.c` należy go skompilować używając kompilatora `cc`. **Nie jest w tym przypadku potrzebny plik *makefile*!** Rozważmy następujący przykład:

```
W katalogu bieżącym mamy tylko dwa pliki źródłowe. Na dysku NIE MA PLIKU
MAKEFILE!
$ ls -l
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello.c
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello2.cpp

Program make uruchamiamy bez żadnych parametrów. Ponieważ na dysku nie ma
pliku makefile generowany jest błąd.
$ make
make: *** No targets specified and no makefile found.  Stop.

Wywołujemy program make a nazwę celu podajemy w wierszu poleceń. Program
make sam rozpoznał z jakim plikiem ma do czynienia (po rozszerzeniu .c) i
sam uruchomił odpowiedni kompilator (tu: cc).
$ make hello
cc     hello.c   -o hello

Upewniamy się, że na dysku powstał plik wykonywalny hello.
$ ls -l
-rwxr-xr-x    1 artur    students     4819 Oct 13 07:52 hello
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello.c
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello2.cpp

Jak wyżej, ale teraz mamy plik źródłowy w języku c++ (rozszerzenie .cpp).
Tym razem program make uruchomił kompilator g++.
$ make hello2
g++     hello2.cpp   -o hello2

Upewniamy się, że na dysku powstał plik wykonywalny hello2.
$ ls -l
-rwxr-xr-x    1 artur    students     4819 Oct 13 07:52 hello
```

```
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello.c
-rwxr-xr-x    1 artur    students     5164 Oct 13 07:53 hello2
-rw-r-----    1 artur    students       82 Oct 11 15:32 hello2.cpp
```