Systemy do budowy oprogramowania

Po co i dlaczego?

- Gdy tworzysz aplikację to:
  - Komplikujesz kod
  - Podłączasz inne biblioteki (proces linkowania)
  - Dystrybuujesz kod źródłowy i/lub binarny
- Ponadto:
  - Testujesz oprogramowanie
  - Testujesz rozpowszechniany pakiet
  - Sprawdzasz rezultaty testów

Kompilacja

- Ręczna:

```
gcc -DMYDEFINE -c myapp.o myapp.cpp
```

- Trudna w wykonaniu gdy:
  - Bardzo dużo plików
  - Część plików winna być komplikowana tylko dla jakieś szczególnej platformy
  - Elementy zależne od platformy, komplilatora

Linkowanie

- Ręczne

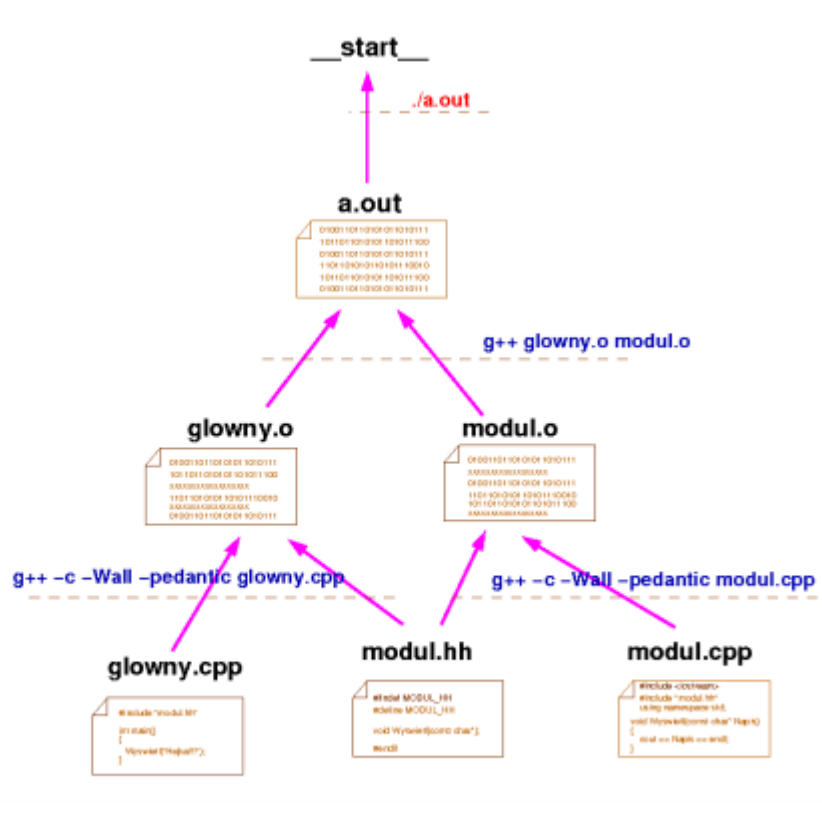
```
ld –o myqpp File1.o File2.o File3.o –lc –lmylib
```

- Znów problem w przypadku dużej ilości plików

Dlaczego automatyzować

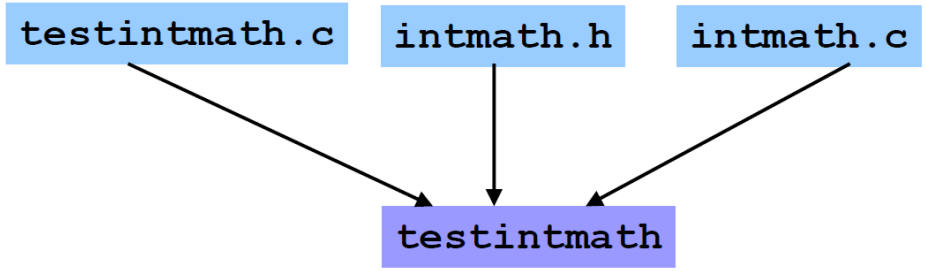
- Podstawowym zadaniem programisty jest rozwój oprogramowania, a nie spędzanie czasu nad żmudnym budowaniem systemu
- Zastanów się „co sprzedajesz”?
  - Symulator lotów
  - Symulator lotów wraz z własnoręcznie napisanym systemy do jego budowy
- Końcowego użytkownika nie interesuje jak system (aplikacja) się buduje interesuje go tylko posiadanie sprawnie działającej aplikacji
- Dlatego należy wykorzystywać system służące do budowania oprogramowanie

```
__start__: a.out
    TAB    ./a.out

a.out: glowny.o modul.o
        g++ glowny.o modul.o

glowny.o: glowny.cpp modul.hh
        g++ -c -Wall -pedantic glowny.cpp

modul.o: modul.hh modul.cpp
        g++ -c -Wall -pedantic modul.cpp
```



Make

- System do automatycznej budowy oprogramowania
- Założenie:
  - Program składa się z 3 plików:
    - intmath.h: zawarty w intmath.c i testintmath.c
    - intmath.c: realizacja funkcji matematycznych
    - testintmath.c: realizacja testów funkcji matematycznych
  - Tworzenie aplikacji testintmath




```
gcc -Wall -ansi -pedantic -o testintmath testintmath.c intmath.c
```

Proces budowy aplkacji

- Poprocesowanie (gcc –E intmath.c > intmath.i)
  - Usuwanie pleceń preprocesora
  - Budowa intmath.i i testintmath.i
- Kompilacja (gcc –S intmath.i)
  - Konwersja do asemblera
  - Budowa intmath.s i testintmath.s
- Skład (gcc –c intmath.s)
  - Konwersja do języka maszynowego
  - Budowa intmath.o i testintmath.o – pliki binarne
- • Linkowanie (gcc –o testintmath testintmath.o intmath.o –lc)
  - Tworzenie pliku wykonywalnego
  - Budowa testintmath jako pliku binarnego

Dlaczego Makefile

- Pisanie na linii poleceń staje się nudne
  - Długie polecenie dla kompilatorem z flagami i nazwami plików
  - Łatwo popełnić błąd
- Każdorazowe kompilowanie wszystkiego od zera jest czasochłonne
  - Powtarzanie wstępnego przetwarzania, kompilacja, składanie i łączenie
  - Powtarzanie tych kroków dla każdego pliku, nawet jeśli tylko jeden się zmienił
- Uniksowe narzędzie Makefile
  - `Makefile`: plik zawierający informacje niezbędne do zbudowania programu
    - Zawiera listę plików oraz zależności
  - Przekompiluj lub ponownie łącz tylko w razie potrzeby
    - Gdy plik zależny został zmieniony od momentu uruchomienia polecenia
    - E.g. jeśli zmienią się intmath.c, zrekompiluj plik intmath.c, ale nie testintmath.c
- Po prostu wpisz "`make`" lub "`make -f (nazwa_pliku)`"

Podstawowe składniki Makefile

- Wpisujemy w poszczególne linii
  - Cel – plik jaki chcemy utworzyć
  - Zależności - pliki od których zależy utworzenie pliku celu
  - Polecenie – jakie musi być wykonane aby plik powstał (zawsze poprzedzone znakiem TAB)

```
testintmath: testintmath.o intmath.o
    gcc -o testintmath testintmath.o intmath.o
```

```
intmath.o: intmath.c intmath.h
    gcc -Wall -ansi -pedantic -c -o intmath.o intmath.c
```

Makefile #1

- Trzy polecenia
  - testintmath: linkowanie testintmath.o i intmath.o
  - testintmath.o: komplikacja testintmath.c, zależy od intmath.h
  - intmath.o: komplikacja intmath.c, zależy od intmath.h

```
testintmath: testintmath.o intmath.o
   gcc -o testintmath testintmath.o intmath.o

testintmath.o: testintmath.c intmath.h
    gcc -Wall -ansi -pedantic -c -o testintmath.o testintmath.c

intmath.o: intmath.c intmath.h
    gcc -Wall -ansi -pedantic -c -o intmath.o intmath.c
```

Cele, które nie są plikami

- Wprowadzenie przydatnych skrótów
  - ‚make all' – buduj cały projekt
  - ‚make clobber' – kasuj pliki tymczasowe, core, binarne etc
  - ‚make clean' – kasuj pliki binarne

```
all: testintmath

clobber: clean
	rm -f *~ \#*\# core

clean:
	rm -f testintmath *.o
```

Makefile #2

```
# Build rules for non-file targets
all: testintmath
clobber: clean
   rm -f *~ \#*\# core
clean:
   rm -f testintmath *.o

# Build rules for file targets
testintmath: testintmath.o intmath.o
   gcc -o testintmath testintmath.o intmath.o
testintmath.o: testintmath.c intmath.h
   gcc -Wall -ansi -pedantic -c -o testintmath.o testintmath.c
intmath.o: intmath.c intmath.h
   gcc -Wall -ansi -pedantic -c -o intmath.o intmath.c
```

Przydatne skróty

- Plik cel $@
- Pierwszy element na liście zależności $<

```
testintmath: testintmath.o intmath.o
    gcc -o testintmath testintmath.o intmath.o
```

```
testintmath: testintmath.o intmath.o
    gcc -o $@ $< intmath.o
```

Makefile #3

```
# Build rules for non-file targets
all: testintmath
clobber: clean
    rm -f *~ \#*\# core
clean:
    rm -f testintmath *.o

# Build rules for file targets
testintmath: testintmath.o intmath.o
   gcc -o $@ $< intmath.o
testintmath.o: testintmath.c intmath.h
    gcc -Wall -ansi -pedantic -c -o $@ $<
intmath.o: intmath.c intmath.h
    gcc -Wall -ansi -pedantic -c -o $@ $<
```

Znak %

- W poprzednim dwa razy powtarzało się:

```
gcc -Wall -ansi -pedantic -c -o $@ $<
```

- Zawsze miało zastosowanie do plików „*.o” i „*.c”

```
%.o: %.c
    gcc -Wall -ansi -pedantic -c -o $@ $<
```

```
testintmath: testintmath.o intmath.o
   gcc -o $@ $< intmath.o
testintmath.o: testintmath.c intmath.h

intmath.o: intmath.c intmath.h
```

Zmienne dla kompilacji i linkowania

- Prosta zmiana kompilatora
  - CC = gcc
  - Używamy jako: $(CC) –o $@ $< intmath.o
- Zmiana opcji kompilatora
  - CFLAGS = -Wall -ansi –pedantic
  - Używamy jako: $(CC) $(CFLAGS) -c -o $@ $<

```
CC = gcc
# CC = gccmemstat

CFLAGS = -Wall -ansi -pedantic
# CFLAGS = -Wall -ansi -pedantic -g
# CFLAGS = -Wall -ansi -pedantic -DNDEBUG
# CFLAGS = -Wall -ansi -pedantic -DNDEBUG -O3
```

Konstrukcje warunkowe

Konstrukcja	Znaczenie
`ifeq`	Sprawdza, czy dwie wartości są sobie równe
`ifneq`	Sprawdza, czy dwie wartości są od siebie różne
`ifdef`	Sprawdza, czy zdefiniowano daną zmienną
`ifndef`	Sprawdza, czy nie zdefiniowano danej zmiennej

Konstrukcje warunkowe

Przykład konstrukcji `if`

```
ifeq ($(shell uname -o), "GNU/Linux")
     CPPFLAGS += -DLINUX
else
     CPPFLAGS += -DOS_UNKNOWN
endif
```

Konstrukcje warunkowe

Testowanie definicji zmiennych

```
ifdef debug
     CFLAGS += -g
else
     CFLAGS += -O2
endif
```

Wywołanie

```
$ make debug=1
```

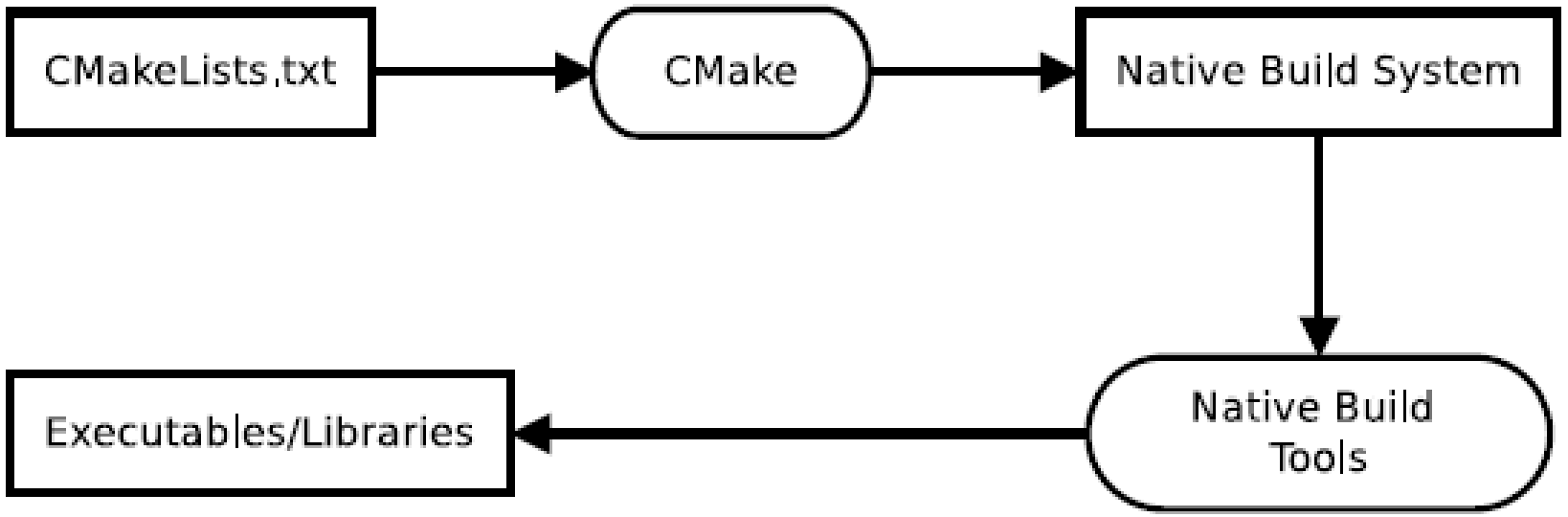
Narzędzie CMake

- Właściwości CMake:
  - Jest wieloplatformowe
  - Wspiera natywny sposób budowy oprogramowania:
    - Linux → makefile
    - Windows → VSProject
    - Apple → XCode
  - Jest darmowe, o otwartym kodzie

Narzędzie CMake

- **Dalsza charakterystyka :**
  - Wykorzystywane w dużych projektach (np. KDE4, VTK)
  - Wsparcie dla makr
  - Moduły do automatycznego wyszukiwania/konfigurowania oprogramowania
  - Własne cele kompilacji lub komendy
  - Uruchamianie zewnętrznych programów
  - Prosta składnia
  - Wsparcie dla wyrażeń regularnych

Działanie CMake



Podstawowe pojęcia CMake

- CmakeLists.txt
  - Plik tekstowy opisujący parametry projektu oraz opisujący przebieg budowy projektu napisany w prostym jeżyku CMake.
- Moduły CMake
  - Specjalne pliki CMake napisane z przeznaczeniem znajdowania specyficznych plików wymaganych przez projekt, np. biblioteki, czy pliki nagłówkowe. Powstały po to, aby wykorzystywać je w wielu projektach. Przykłady: FindJava.cmake, FindZLIB.cmake, FindQt4.cmake.

Podstawowe pojęcia CMake

- Source Tree zawiera:
  - pliki wejściowe CMake (CmakeLists.txt)
  - pliki źródłowe projektu (hello.cpp)
  - pliki nagłówkowe projektu (hello.h)
- Binary Tree zawiera:
  - natywne pliki budowy projektu (makefiles)
  - efekty procesu budowy: biblioteki, pliki wykonywalne i inne
- Source Tree i Binary Tree:
  - mogą być w jednym katalogu nadrzędnym (In-source)
  - mogą być w różnych katalogach nadrzędnych (Out-of-source)

Podstawowe pojęcia CMake

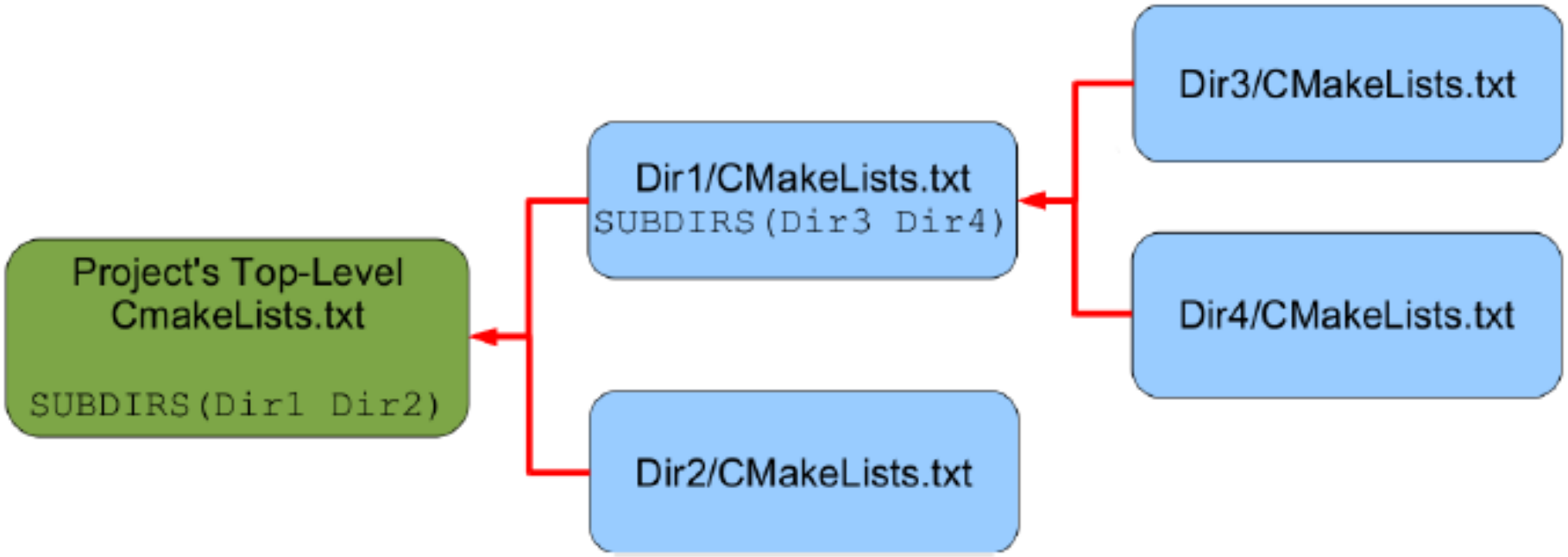
- CMAKE_MODULE_PATH
  - Ścieżka do modułów CMake
- CMAKE_INSTALL_PREFIX
  - Ścieżka do umiejscowienia plików po wywołaniu make install
- CMAKE_BUILD_TYPE
  - Typ budowy (Debug, Release, ...)
- BUILD_SHARED_LIBS
  - Przełącznik pomiędzy bibliotekami współdzielonymi i statycznymi
- Zmienne mogą być definiowane w pliku CmakeLists.txt lub jako argument polecenia cmake, np.:

cmake -DBUILD SHARED LIBS=OFF lub przez GUI: ccmake

CMake cache

- Utrzymywany w Build Tree (CMakeCache.txt)
- Zawiera wpisy typy VAR:TYPE=VALUE
- Zapełniany/aktualizowany podczas fazy konfiguracji
- Przyspiesza proces budowy
- Może być zainicjalizowany poleceniem cmake -C <plik>
- Z reguły nie ma potrzeby ręcznej edycji
- Można zmieniać wartości z poziomu GUI

Struktura plików Source Tree

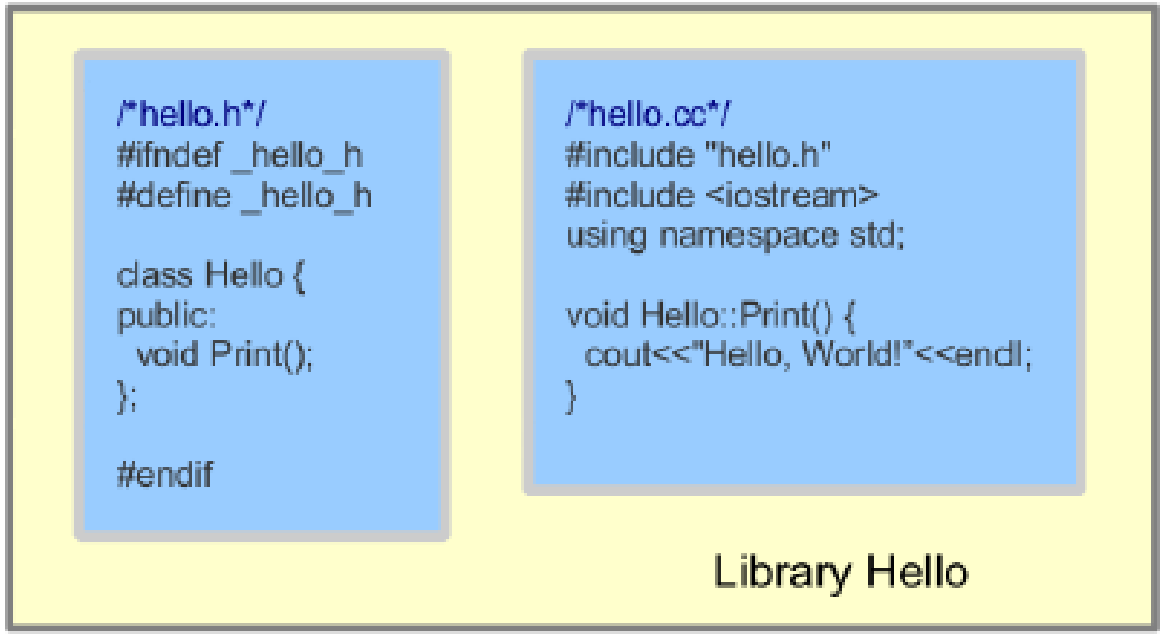


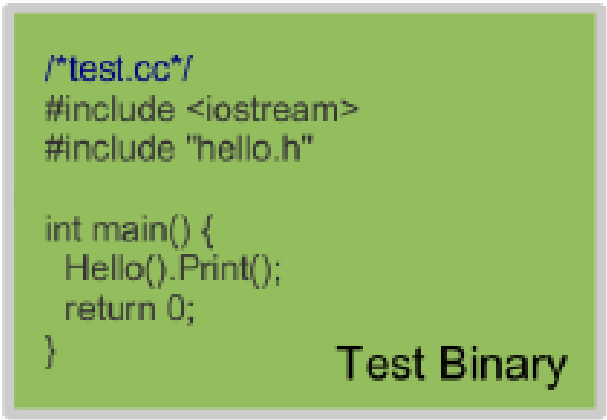
Uzycie CMake

- utworzenie katalogu budowy projektu
  - mkdir build; cd build
- konfiguracja
  - cmake [opcje] <source tree>
- budowa
  - make
- instalacja
  - make install

Hello World w CMake

- **Katalog projektu**
- CMakeLists.txt
- podkatalog Hello:
  - CMakeLists.txt
  - hello.h
  - hello.cc
- podkatalog Test:
  - CMakeLists.txt
  - test.cc





Hello World w CMake

```
# CmakeLists.txt z katalogu głównego
    PROJECT( HELLO )
    ADD_SUBDIRECTORY( Hello )
    ADD_SUBDIRECTORY( Test )

# CmakeLists.txt z katalogu Hello
# Dodaje bibliotekę o nazwie Hello (libHello.a
# pod Linuksem) na podstawie pliku źródłowego hello.cc
    ADD_LIBRARY( Hello hello )
```

Hello World w CMake

```
# CmakeLists.txt z katalogu Test
# Upewnienie się, ze kompilator znajdzie pliki
# nagłówkowe z katalogu Hello
    INCLUDE_DIRECTORIES(${HELLO_SOURCE_DIR}/Hello)
# Dodanie pliku wykonywalnego "helloWorld",
# który tworzony jest na podstawie pliku
# źródłowego "test.cc". Odpowiednia końcówka
# jest dodawana automatycznie.
    ADD_EXECUTABLE(helloWorld test)
# Linkowanie biblioteki Hello z plikiem wykonywalnym
    TARGET_LINK_LIBRARIES(helloWorld Hello)
```

Składnia plików CmakeLists.txt

- Komentarz: # to jest komentarz
- wywołanie polecenia: COMMAND(arg1 arg2 ...)
- listy wartości: A; B; C
- zmienne ${VAR}
- instrukcje warunkowe:
  - IF() ... ELSE()/ELSEIF() ... ENDIF()
  - WHILE() ... ENDWHILE()
  - FOREACH() ... ENDFOREACH()
- I wyrażenia regularne

Składnia plików CmakeLists.txt

- Polecenia:
  - INCLUDE_DIRECTORIES(”dir1” ”dir2” ...)
  - AUX_SOURCE_DIRECTORY(”source”)
  - ADD_EXECUTABLE
  - ADD_LIBRARY
  - ADD_CUSTOM_TARGET
  - ADD_DEPENDENCIES(target1 t2 t3)
  - ADD_DEFINITIONS(”-Wall -ansi -pedantic”)
  - TARGET_LINK_LIBRARIES(target-name lib1 lib2 ...)
  - SET_TARGET_PROPERTIES(...) przykładowe własciwosci:
  - OUTPUT_NAME, VERSION, ...
  - MESSAGE(STATUS, FATAL ERROR, ... ”message”)
  - INSTALL(FILES ”f1” ”f2” ”f3” DESTINATION .)
  - FIND_FILE
  - FIND_LIBRARY

Podsumowanie CMake

- Zawartość pliku CMakeLists.txt

```
PROJECT(MyProject C)
ADD_LIBRARY(MyLibrary STATIC libSource.c)
ADD_EXECUTABLE(MyProgram main.c)
TARGET_LINK_LIBRARIES(MyProgram MyLibrary)
```

- Użycie programu CMake

```
$ cmake sciezka_do_zrodel
$ make
```