Wydział Elektroniki,
Fotoniki i Mikrosystemów

Inżynieria Oprogramowania dla Elektromobilności

Ćwiczenie nr 1. Zajęcia wprowadzające. BHP i zapoznanie z narzędziami

Zagadnienia do opracowania:

- zadania preprocesora, kompilatora i konsolidatora w procesie budowania aplikacji
- kompilacja i asemblacja kodu źródłowego
- system budowania CMake

Literatura

[1] *CMake Tutorial*. `https://cmake.org/cmake/help/latest/guide/tutorial/index.html`. [Online; dostęp 03.10.2022].

[2] Kornelia Indykiewicz. *Wykład: Inżynieria Oprogramowania dla Elektromobilności*. 2022.

Spis treści

1. Cel ćwiczenia

Celem ćwiczenia jest zapoznanie z zasadami BHP w laboratorium komputerowym oraz podstawowymi narzędziami wykorzystywanymi w trakcie realizacji kursu.

2. Wprowadzenie

2.1. System budowania CMake

CMake (skrót od ang. *Cross–platform Make*) to narzędzie służące do automatyzacji procesu budowania aplikacji napisanych w języku *C* lub *C++*. Jest on niezależny od platformy sprzętowej, systemu operacyjnego i kompilatora. Wspiera również kompilację skrośną (*ang. cross–compilation*). ***CMake*** bazuje na języku skryptowym, który umożliwia konfigurację procesu kompilacji i konsolidacji kodu źródłowego. Kod skryptów umieszczany jest w plikach o nazwie ***CMakeLists.txt***.

Na listingu 1. przedstawiono prosty skrypt systemu budowania ***CMake*** zawierający trzy instrukcje. Jego wykonanie umożliwia zbudowanie aplikacji napisanej w języku *C++* składającej się z jednej jednostki translacji o nazwie *main.cpp*. Pierwsze polecenie, tj. ***cmake_minimum_required()*** określa minimalną wersję systemu budowania ***CMake***, jaka jest wymagana do wykonania skryptu (tu: wersja 3.22). Instrukcja ***project()*** ustawia nazwę projektu (pod zmienną *PROJECT_NAME*). Instrukcja ***add_executable()*** określa jakie pliki będą współtworzyć plik wykonywalny aplikacji (tu: plik *main.cpp* jest jedynym źródłem tworzącym plik wykonywalny *my_app*). Skrypty systemu ***CMake*** są niewrażliwe na wielkość liter. Aby wykonać skrypt zawarty w pliku ***CMakeLists.txt*** należy za pomocą konsoli systemowej wywołać polecenie:

cmake [ścieżka_do_pliku_CMakeLists.txt]

```
cmake_minimum_required(VERSION 3.22)
project(my_app)

add_executable(my_app main.cpp)
```

Listing 1. Podstawowy skrypt *CMake*

Jeżeli polecenie wywoływane jest z poziomu katalogu zawierającego plik ***CMakeLists.txt***, to można posłużyć się uproszczonym zapisem:

cmake .

Pojedyncza kropka (***.***) określa bieżący katalog w systemie plików, podczas gdy podwójna kropka (***..***) określa katalog nadrzędny (*ang. parent directory*). W wyniku wywołania polecenia ***cmake*** zostanie wygenerowany plik ***Makefile***, zawierający zestaw instrukcji narzędzia ***GNU Make***, służącego do budowania pliku wykonywalnego aplikacji. Wywołanie polecenia ***make*** z poziomu katalogu zawierającego plik ***Makefile*** poskutkuje utworzeniem pliku wykonywalnego aplikacji (tu: *my_app*). Polecenie ***make*** nie wymaga przekazywania dodatkowych argumentów. Poza plikiem ***Makefile*** polecenie ***cmake*** generuje kilka dodatkowych plików (*CMakeCache.txt*, *cmake_intall.cmake*) oraz katalog *CMakeFiles*, co może zaciemniać wprowadzoną strukturę katalogów projektu. Powszechnym rozwiązaniem jest wstępne utworzenie pustego katalogu pomocniczego ***build*** jako folderu przeznaczonego na przechowywanie plików wygenerowanych w procesie przetwarzania skryptu ***CMake***. Wówczas, wywołanie polecenia:

cmake ..

z poziomu katalogu ***build*** poskutkuje umieszczeniem wygenerowanych plików pomocniczych wewnątrz katalogu ***build***.

Częstym wymaganiem jest ustawienie określonej wersji standardu języka *C++*. W tym celu można się posłużyć poleceniem ***set()***, jak na listingu 2:

```
cmake_minimum_required(VERSION 3.22)
project(my_app)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)

add_executable(my_app main.cpp)
```

Listing 2. Ustawianie standardu języka *C++* w skrypcie *CMake*

Instrukcja ***set(CMAKE_CXX_STANDARD 20)*** przypisuje zmiennej *CXX_STANDARD* wartość 20, co przekłada się na użycie flagi ***−std=c++20*** (lub równoważnej) w procesie kompilacji kodu źródłowego. Z kolei instrukcja ***set(CMAKE_CXX_STANDARD_REQUIRED TRUE)*** określa, że proces budowania pliku wykonywalnego aplikacji ma zostać przerwany, jeżeli zainstalowany kompilator nie wspiera ustalonego standardu języka *C++*. W przeciwnym razie standard języka *C++* zostałby obniżony do najaktualniejszej wersji wspieranej przez wykorzystywany kompilator. W analogiczny sposób, korzystając z instrukcji ***set()***, można zwiększyć poziom szczegółowości logowania podczas budowania pliku wykonywalnego aplikacji (listing 3).

```
cmake_minimum_required(VERSION 3.22)
project(my_app)

set(CMAKE_VERBOSE_MAKEFILE ON)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)

add_executable(my_app main.cpp)
```

Listing 3. Zwiększenie poziomu szczegółowości logowania narzędzia ***Make***

Skrypty systemu ***CMake*** pozwalają również na definiowanie własnych zmiennych pomocniczych. Służy do tego opisane wcześniej polecenie ***set()***. Przykład przedstawiono na listigu 4. Zmienna pomocnicza *SOURCE_FILES* przechowuje nazwy wszystkich plików źródłowych wykorzystywanych w procesie budowania pliku wykonywalnego aplikacji (tu: jeden plik – *main.cpp*). Wyłuskanie wartości zmiennej realizowane jest za pomocą operatora ${}. Linie poprzedzone symbolem # traktowane są jako komentarze. Wszystkie instrukcje ***set()*** muszą zostać umieszczone powyżej instrukcji ***add_executable()***.

```
1  cmake_minimum_required(VERSION 3.22)
2  project(my_app)
3
4  set(CMAKE_VERBOSE_MAKEFILE ON)
5
6  set(CMAKE_CXX_STANDARD 20)
7  set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
8
9  #Definicja zmiennej pomocniczej SOURCE_FILES
10 set(SOURCE_FILES main.cpp)
11
12 add_executable(my_app ${SOURCE_FILES})
```

Listing 4. Definiowanie zmiennych pomocniczych w skrypcie *CMake*

Skrypt z listingu 4. można w łatwy sposób rozbudować, aby uwzględnić dodatkowe pliki (nagłówkowe i źródłowe) wchodzące w skład projektu. Zostało to przedstawione na listingu 5. Oprócz pliku *main.cpp* plik wykonywalny aplikacji współtworzą teraz pliki nagłówkowe *definitions.h* i *paraser.h* oraz plik źródłowy *parser.cpp*. Aby uniknąć dublowania kodu, nazwa pliku wykonywalnego została przekazana do instrukcji ***add_executable()*** za pośrednictwem zmiennej *PROJECT_NAME* (ustawianej za pomocą polecenia ***project()***).

```
cmake_minimum_required(VERSION 3.22)
project(my_app)

set(CMAKE_VERBOSE_MAKEFILE ON)

set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)

#Definicja zmiennej pomocniczej HEADER_FILES
set(HEADER_FILES
    definitions.h
    parser.h)

#Definicja zmiennej pomocniczej SOURCE_FILES
set(SOURCE_FILES
    main.cpp
    parser.cpp)

add_executable(${PROJECT_NAME} ${SOURCE_FILES} ${
   HEADER_FILES})
```

Listing 5. Dodawanie kolejnych plików projektu w skrypcie *CMake*

Jeżeli kod źródłowy projektu ma zostać skonsolidowany do postaci biblioteki programistycznej należy posłużyć się poleceniem ***add_library()*** zamiast polecenia ***add_executable()***. Instrukcja ***add_library()*** przyjmuje dodatkowy parametr określający czy kod ma zostać przetworzony do postaci biblioteki statycznej:

add_library(${PROJECT_NAME} STATIC ${SOURCE_FILES} ${HEADER_FILES})

czy dynamicznej:

add_library(${PROJECT_NAME} SHARED ${SOURCE_FILES} ${HEADER_FILES}).

System ***CMake*** umożliwia również wskazanie ścieżki, pod którą ma zostać umieszczony wygenerowany plik (wykonywalny bądź biblioteki). W tym celu należy za pomocą instrukcji ***set()*** ustawić wartości odpowiednich zmiennych:

- ***CMAKE_ARCHIVE_OUTPUT_DIRECTORY*** – w przypadku budowania pliku biblioteki statycznej;
- ***CMAKE_LIBRARY_OUTPUT_DIRECTORY*** – w przypadku budowania pliku biblioteki dynamicznej;
- ***CMAKE_RUNTIME_OUTPUT_DIRECTORY*** – w przypadku budowania pliku wykonywalnego aplikacji.

Przykład przedstawiono na listingu 6. Plik wykonywalny *my_app* zostanie umieszczony w katalogu *deploy*. Jeżeli katalog nie istnieje, to wywołanie polecenia ***make*** poskutkuje jego utworzeniem. Zmienna ***PROJECT_BINARY_DIR*** domyślnie przechowuje ścieżkę do folderu, w którym generowane są pliki ***CMake*** (np. do folderu ***build***).

```
1  cmake_minimum_required(VERSION 3.22)
2  project(my_app)
3
4  set(CMAKE_VERBOSE_MAKEFILE ON)
5
6  set(CMAKE_CXX_STANDARD 20)
7  set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
8
9  #Definicja zmiennej pomocniczej HEADER_FILES
10 set(HEADER_FILES
11     definitions.h
12     parser.h)
```

```
13
14 #Definicja zmiennej pomocniczej SOURCE_FILES
15 set(SOURCE_FILES
16     main.cpp
17     parser.cpp)
18
19 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${
   PROJECT_BINARY_DIR}/deploy)
20
21 add_executable(${PROJECT_NAME} ${SOURCE_FILES} ${
   HEADER_FILES})
```

Listing 6. Określanie położenia budowanego pliku wykonywalnego w skrypcie *CMake*

Aby skonsolidować bibliotekę programistyczną podczas budowania pliku wykonywalnego aplikacji należy wywołać polecenie ***target_link_libraries()*** po instrukcji ***add_executable()***. Przykład przedstawiono na listingu 7. Aplikacja *my_app* konsoliduje bibliotekę *my_lib*. Pierwszy argument wywołania instrukcji ***target_link_libraries()*** stanowi nazwa budowanego pliku wykonywalnego. Drugi argument to pełna ścieżka do konsolidowanej biblioteki programistycznej. Polecenie ***find_library()*** umożliwia automatyczne wyszukanie określonej biblioteki programistycznej (tu: *my_lib*) i zapisanie ścieżki do pliku pod wskazaną zmienną pomocniczą (tu: *LIB_FILE*). Flaga ***REQUIRED*** określa, że proces generowania pliku ***Makefile*** ma zostać przerwany w przypadku nieodnalezienia pliku biblioteki. Jeżeli biblioteka leży w niestandardowej lokalizacji na dysku komputera, koniecznym może okazać się jawne wskazanie katalogów do przeszukania przez system ***CMake***. Służy do tego opcja ***HINTS***:

find_library([nazwa_zmiennej_pomocniczej] [nazwa_biblioteki]
HINTS [bezwzględna_ścieżka_do_katalogu_biblioteki]
REQUIRED)

Jeżeli załączane pliki nagłówkowe położone są w innym katalogu niż plik ***CMakeList.txt*** (tu: w katalogu *include*), to warto posłużyć się poleceniem ***target_link_directories()*** w celu poinformowania kompilatora o niestandardowej lokalizacji plików nagłówkowych. W przeciwnym razie koniecznym będzie uwzględnianie ścieżki do pliku nagłówkowego w dyrektywach preprocesora #include. Flaga ***PUBLIC*** określa, że wskazany katalog zawiera publiczny interfejs aplikacji lub biblioteki.

```
1  cmake_minimum_required(VERSION 3.22)
2  project(my_app)
3
4  set(CMAKE_VERBOSE_MAKEFILE ON)
5
6  set(CMAKE_CXX_STANDARD 20)
7  set(CMAKE_CXX_STANDARD_REQUIRED TRUE)
8
9  #Definicja zmiennej pomocniczej HEADER_FILES
10 set(HEADER_FILES
11     include/definitions.h
12     include/parser.h)
13
14 #Definicja zmiennej pomocniczej SOURCE_FILES
15 set(SOURCE_FILES
16     main.cpp
17     parser.cpp)
18
19 set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${
       PROJECT_BINARY_DIR}/deploy)
20
21 find_library(LIB_FILE my_lib REQUIRED)
22
```

```
23 add_executable(${PROJECT_NAME} ${SOURCE_FILES} ${
     HEADER_FILES})
24 target_link_libraries(${PROJECT_NAME} ${LIB_FILE})
25 target_link_directories(${PROJECT_NAME} PUBLIC
     include)
```

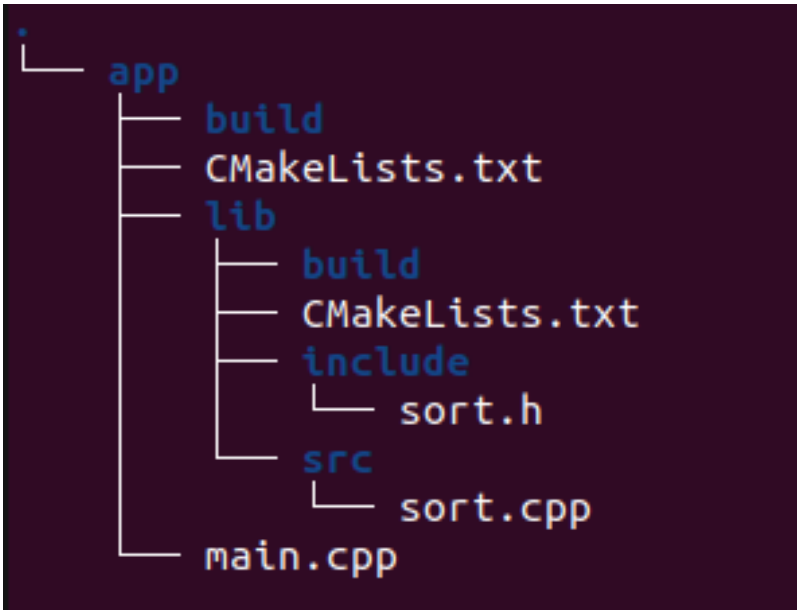
Listing 7. Konsolidacja biblioteki programistycznej w skrypcie *CMake*

Aby zapewnić przenośność kodu między odrębnymi platformami sprzętowymi i systemami operacyjnymi należy unikać jawnego podawania bezwzględnych ścieżek do plików bądź katalogów w skryptach ***CMake***. Wygodnym rozwiązaniem jest odwoływanie się do zmiennych zawierających adresy katalogów powiązanych z systemem budowania ***CMake*** i tworzenie przy ich pomocy ścieżek względnych. Jedną z takich zmiennych jest wspomniana wcześniej ***PROJECT_BINARY_DIR***, odnosząca się do folderu z plikami wygenerowanymi przez system ***CMake*** (porównaj: listing 6. *${PROJECT_BINARY_DIR}/deploy*). Inną przydatną zmienną jest ***CMAKE_CURRENT_SOURCE_DIR***, zawierająca ścieżkę do aktualnie wykonywanego pliku ***CMakeLists.txt***.

3. Program ćwiczenia

Zadanie

Dany jest projekt o strukturze katalogów jak na rys. 3.1. Projekt składa się z aplikacji o nazwie ***sort_app*** oraz biblioteki dynamicznej o nazwie ***libsort***, zaimplementowanych w języku *C++*. Plik wykonywalny aplikacji tworzy jeden plik źródłowy – ***main.cpp*** oraz dynamicznie konsolidowana biblioteka ***libsort.so***. Plik nagłówkowy (***sort.h***) stanowi interfejs biblioteki ***libsort***, natomiast plik źródłowy (***sort.cpp***) zawiera jej implementację. Biblioteka udostępnia trzy funkcje realizujące odpowiednie algorytmy sortowania: ***bubbleSort()***, ***quickSort()*** oraz ***mergeSort()***. Katalogi ***build*** są przeznaczone na organizację plików wygenerowanych przez system budowania ***CMake*** w wyniku przetworzenia odpowiednich skryptów ***CMakeLists.txt***.



Rys. 3.1. Struktura katalogów projektu

Korzystając z narzędzia ***CMake*** napisz skrypty, za pomocą których zbudujesz bibliotekę ***libsort*** oraz aplikację ***sort_app***. Kod umieść w przygotowanych plikach ***CMakeLists.txt***. Przyjmij, że wymaganą wersją systemu

CMake jest wersja 3.22, a wymaganym standardem języka jest *C++20*. Skonsolidowane pliki wynikowe powinny zostać umieszczone w sposób automatyczny w katalogach ***deploy*** wewnątrz dedykowanych katalogów ***build***.