NARZĘDZIA #3

SYSTEMY BUDOWANIA

CODERS SCHOOL

MATEUSZ ADAMSKI

ŁUKASZ ZIOBROŃ

AGENDA

1. `make`
2. `cmake`

ZADANIA

Repo GH `coders-school/cmake`

https://github.com/coders-school/cmake/

KILKA PYTAŃ

- jak współpraca grupowa?
- kto zrozumiał polimorfizm?

DALSZY ROZWÓJ SHM

Zachęcamy wszystkich do ciągłego rozwijania projektu SHM "w tle", uzupełniając go o rzeczy, których na bieżąco będziemy się uczyć w trakcie kursu, takie jak:

- system budowania
- testy jednostkowe
- nowości z C++11/14/17/20

Nie będziemy za to przyznawać dodatkowych punktów ani też sprawdzać tego kodu. Sami rozwijajcie tę aplikację i dzielcie się spostrzeżeniami wewnątrz swojej grupy i z innymi grupami na Discordzie.

Celem jest skuteczna nauka C++ i szybkie stosowanie nowo poznanych rzeczy w projektach, dzięki czemu zapamiętacie je lepiej :)

make

CODERS SCHOOL

PLIKI MAKEFILE

STRUKTURA PLIKU MAKEFILE

```
VARIABLE = value

targetA: dependencyA1 dependencyA2
[TAB] command $(VARIABLE)

targetB: dependencyB1
[TAB] command
```

PRZYKŁAD - GENEROWANIE PREZENTACJI W LATEX

```
TEX = pdflatex -shell-escape -interaction=nonstopmode -file-line-error
MAKE = make
CODE_DIR = src

.PHONY: all view

all: calculator pdf

view:
    evince ContractProgramming.pdf

pdf: ContractProgramming.tex
    $(TEX) ContractProgramming.tex

calculator:
    $(MAKE) -C $(CODE_DIR)
```

KOMPILACJA W C++

PAMIĘTACIE FAZY KOMPILACJI?

```
SOURCES=$(wildcard src/*.cpp)
OBJECTS=$(patsubst %.cpp, %.o, $(SOURCES))

main: $(OBJECTS)
    g++ $^ -o $@

$(OBJECTS): src/%.o : src/%.cpp src/%.hpp
    g++ -c $< -o $@
```

ZMIENNE PAMIĘTAJĄCE KONTEKST

- `$@` - nazwa pliku targetu w aktualnie uruchomionej regule
- `$<` - nazwa pierwszej zależności
- `$^` - lista wszystkich zależności (zawiera ewentualne duplikaty)
- `$?` - lista wszystkich zależności, które są nowsze niż target

ZADANIE

W katalogu greeter znajdziesz malutki program. Zapoznaj się z jego kodem.

- Skompiluj program z linii komend i uruchom go.
- Napisz prosty Makefile dla tego programu. Zbuduj go za pomocą `make` i uruchom.

ZAKLĘCIE KOMPILACJI

```
g++ -std=c++17 -Wall -Werror -Wextra -pedantic *.cpp - o greeter
./greeter
```

POLECENIE `make`

- domyślnie szuka w bieżącym katalogu pliku Makefile
- automatyzuje czynności poprzez wykonywanie receptur zapisanych w plikach Makefile
- domyślnie wykonuje pierwszą recepturę
- pozwala na warunkowe wykonywanie czynności
- pozwala definiować wiele zależności
- domyślnie uwzględnia daty modyfikacji zależności i na tej podstawie podejmuje decyzję, czy wykonać daną recepturę

Q&A

LINKI

cpp-polska.pl

cmake

CODERS SCHOOL

CMAKE

- automatyzuje proces budowania dla C/C++
- obsługuje generowanie projektów dla wielu IDE
- może składać się z wielu modułów, które się łączy (odpowiednik #include)
- niezależny od platformy (jeśli jest dobrze napisany)
- konfiguracja budowania w pliku CMakeLists.txt
- generuje system budowania (np. pliki Makefile)

MINIMALNY `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.10)
project(ProjectName)

add_executable(execName main.cpp file.cpp)
```

BUDOWANIE ZA POMOCĄ CMAKE'A

```
mkdir build      # tworzymy katalog z wynikami budowania
cd build         # wchodzimy do tego katalogu
cmake ..         # generujemy system budowania podając ścieżkę do pliku CMakeLi
cmake --build    # budujemy projekt
```

`cmake --build` można zamienić na `make` jeśli wiemy, że na pewno generujemy Makefile.

`cmake --build` jest uniwersalne.

ZADANIE

Napisz prosty CMakeLists.txt dla programu z katalogu `greeter`, zbuduj go za pomocą `cmake` i uruchom.

Q&A

CMAKE - ZMIENNE

Zmienne tworzymy za pomocą komendy `set`

```
set(VARIABLE value)  # Konwencja - UPPERCASE_WITH_UNDERSCORE
```

Przykładowo

```
set(NAME TheGreatestProject)
```

Odnosimy się do nich później obejmując w nawiasy `{}` i poprzedzając znakiem `$`

```
add_executable(${NAME} main.cpp)
```

Spowoduje to utworzenie targetu `TheGreatestProject`, w ramach którego skompilowany zostanie plik `main.cpp`

CMAKE - PREDEFINIOWANE ZMIENNE

CMake domyślnie dostarcza kilka zmiennych. Odwoływanie się do nich bezpośrednio lub ich modyfikacja zazwyczaj nie są uznawane za dobre praktyki.

Możemy za to bez większych problemów wykorzystać zmienną `${PROJECT_NAME}`. Zawiera ona nazwę projektu zdefiniowaną przez komendę `project()`

```
project(vectorFunctions)
add_executable(${PROJECT_NAME} main.cpp vectorFunctions.cpp)
```

TWORZENIE APLIKACJI I BIBLIOTEK

CMake manual

Poniższe komendy możesz potraktować jako "konstruktory". Tworzą one "targety".

```
add_executable(<name> [source1] [source2 ...])
```

```
add_library(<name> [STATIC | SHARED | MODULE] [source1] [source2 ...])
```

```
add_library(${PROJECT_NAME}-lib STATIC functions.cpp modules.cpp)
add_executable(${PROJECT_NAME} main.cpp functions.cpp modules.cpp)
add_executable(${PROJECT_NAME}-ut test.cpp functions.cpp modules.cpp)
```

PROBLEM

Powielona lista plików w różnych "targetach"

PROBLEM #1

Powielona lista plików w różnych "targetach"

ROZWIĄZANIE

Wrzucenie listy plików do zmiennej

ZADANIE

Wrzuć listę plików do zmiennej i skorzystaj z niej

PROBLEM #2

Drobne różnice w plikach pomiędzy targetami

ROZWIĄZANIE

Utworzenie biblioteki

BIBLIOTEKI

Moja definicja biblioteki - zlepek wielu plików cpp bez funkcji `main()`. Biblioteki nie można z tego powodu uruchomić.

ANALOGIA DO PROGRAMOWANIA OBIEKTOWEGO

- Biblioteka = klasa (bazowa)
  - pola, metody = pliki cpp
- Binarka = klasa pochodna
  - finalna, nie można po niej dziedziczyć
- Linkowanie = dziedziczenie
  - zlinkowanie binarki z biblioteką oznacza dorzucenie do niej kodu z biblioteki
  - biblioteki można ze sobą wzajemnie linkować

LINKOWANIE BIBLIOTEK

```
target_link_libraries(<target> ... <item>... ...)
```

```
add_library(lib STATIC functions.cpp modules.cpp)
add_executable(main main.cpp)
add_executable(ut tests.cpp)
target_link_libraries(main lib)
target_link_libraries(ut lib)
```

ZADANIE

Utwórz bibliotekę, która będzie zawierać powtarzające się pliki cpp i zlinkuj z nią targety, które ich używały.

FLAGI KOMPILACJI

```
target_compile_options(<target> [BEFORE]
                               <INTERFACE|PUBLIC|PRIVATE> [items1...]
                               [<INTERFACE|PUBLIC|PRIVATE> [items2...]
                               ...])
```

```
add_executable(${PROJECT_NAME} main.cpp)
target_compile_options(${PROJECT_NAME} PRIVATE -Wall -Wextra -Werror)
```

ZADANIE

Dodaj flagi kompilacji `-Wall -Wextra -Werror -pedantic -Wconversion -O3` do projektu greeter

WŁĄCZANIE STANDARDU C++17

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

Powyższe może nie działać dla MSVC.

```
set_target_properties(${PROJECT_NAME} PROPERTIES
                      CXX_STANDARD 17
                      CXX_STANDARD_REQUIRED ON)
```

```
add_executable(${PROJECT_NAME} main.cpp)
target_compile_features(${PROJECT_NAME} PRIVATE cxx_std_17)
```

ZADANIE

Włącz standard C++17 w projekcie greeter

DODAWANIE TESTÓW DO `ctest`

```
enable_testing()
add_test(NAME <name> COMMAND <command> [<arg>...])
```

```
enable_testing()
add_test(NAME someTests COMMAND ${PROJECT_NAME}-ut)
```

ZADANIE

Dodaj binarkę z testami, która powinna być odpalana za pomocą `ctest`

BUDOWANIE W TRYBIE DEBUG

Domyślnie budowany jest tryb "Release" (bez symboli debugowania)

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Jeśli chcemy wspierać budowanie w trybach Debug i Release powinniśmy mieć do nich oddzielne katalogi z rezultatami budowania

```
mkdir buildDebug
cd buildDebug
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake --build
```

LINKI DLA POSZERZENIA WIEDZY

- CMake - from zero to something - prezentacja z Wro.cpp
- 19 reasons why CMake is actually awesome
- Modern CMake is like inheritance
- CMake basics

Q&A

NARZĘDZIA #3

PODSUMOWANIE

CODERS SCHOOL

CO PAMIĘTASZ Z DZISIAJ?

NAPISZ NA CZACIE JAK NAJWIĘCEJ HASEŁ

PRE-WORK

- Poczytajcie o zasadach SOLID, dotyczących pisania dobrego kodu obiektowego
- Poczytajcie o zasadach dobrego kodu w C++ na CppCoreGuidelines
- Dowiedzcie się czym jest problem diamentowy na Obiektowość #4

POST-WORK

- Dorzućcie do projektu SHM system budowania `cmake` (10 punktów, 2 za każdy podpunkt)
- Przygotujcie SHM do testowania (10 punktów, 2 za każdy podpunkt)

BONUS

- Dostarczenie przed niedzielą 05.07.2020 23:59 (2 punkty za zadanie, razem 4)

`cmake` W SHM

- Użyjcie zmiennej `${PROJECT_NAME}`
- Lista plików cpp w zmiennej
- Wszystko poza plikiem `main.cpp` powinno kompilować się do biblioteki statycznej
- Binarka (main.cpp) powinna linkować się z powyższą biblioteką.
- Napiszcie odpowiedni plik `.github/workflows/main.yml` który spowoduje, że GitHub będzie automatycznie uruchamiał kompilację projektu dla każdego nowego commita.

PODWALINY POD TESTY W SHM

Na podstawie lektury plików CMakeLists.txt z prac domowych wywnioskujcie, w jaki sposób dodawana jest biblioteka `gtest` do testów

- Utwórzcie proste testy do projektu SHM (co najmniej 1 test metodą Copy&Paste z innych prac domowych)
- Skopiujcie odpowiednie pliki, które pozwolą na użycie `gtesta`
- Dodajcie binarkę z testami co CMakeLists.txt. Nazwijcie ją `${PROJECT_NAME}-ut`
- Dodajcie odpalanie testów za pomocą `ctest`
- Zmodyfikujcie plik `.github/workflows/main.yml` aby GitHub dodatkowo uruchamiał jeszcze testy

CODERS SCHOOL