TiDA 05[amb] – Kompilacja z linii komend + Python

Cel zajęć:

- Nauczyć się kompilować kod C++ z linii komend z użyciem różnych parametrów
- Zautomatyzować ten proces w Pythonie (łącznie z ustawieniem środowiska Visual Studio)
- Poznać podstawy testów automatycznych i CI (Continuous Integration).

Przygotowanie projektu

Struktura katalogu:

```
blok04_ambitny/
├─ main.cpp
├─ build_and_run.py
└─ logs/
```

- Folder `logs` – na pliki `.log` z wynikami.
- Plik `main.cpp` – nasz kod testowy.
- Plik `build_and_run.py` – automatyzacja kompilacji, uruchomienia i logowania.

Kod źródłowy `main.cpp`

Program wypisuje informacje o architekturze, trybie kompilacji i czasie kompilacji.

```cpp
#include <iostream>
using namespace std;

#ifndef BUILD_MODE
#define BUILD_MODE "(nieznany tryb)"
#endif

int main() {
#ifdef _M_X64
    cout << "ARCH: x64" << endl;
#elif defined(_M_IX86)
    cout << "ARCH: x86" << endl;
#else
    cout << "ARCH: unknown" << endl;
#endif

    cout << "BUILD_MODE: " << BUILD_MODE << endl;
    cout << "DATE: " << __DATE__ << " TIME: " << __TIME__ << endl;
    cout << "Hello from advanced build system!" << endl;
    return 0;
}
```

💻 Uruchomienie kompilacji z linii komend

Z Developer Command Prompt (x64):

```
cl main.cpp /O2 /MD /D BUILD_MODE="Release_O2_MD_x64" /Feprogram.exe
program.exe
```

Wynik:

```
ARCH: x64
BUILD_MODE: Release_O2_MD_x64
DATE: ...
TIME: ...
Hello from advanced build system!
```

💻 Znaczenie parametrów kompilacji

Parametr	Znaczenie
`/O2`	Optymalizacja szybkości (Release)
`/Ox`	Maksymalna optymalizacja (łączenie strategii)
`/Od`	Bez optymalizacji (Debug)
`/MD`	Łączenie dynamiczne z biblioteką runtime (DLL)
`/MT`	Statyczne łączenie z biblioteką runtime
`/MDd`	Debug + DLL
`/MTd`	Debug + statyczne
`/arch:AVX2`	Użycie instrukcji AVX2 (dla nowoczesnych CPU)

💻 Skrypt `build_and_run.py` – automatyzacja kompilacji i testu

Skrypt:

- sam ustawia środowisko (uruchamia `vcvars64.bat`),
- buduje program z parametrami,
- uruchamia wynikowy `.exe`,
- zapisuje wynik do katalogu `logs`.

```
import subprocess, datetime, os, time
from pathlib import Path

# ======================
# KONFIGURACJA UŻYTKOWNIKA
# ======================
VCVARS = r"C:\Program Files\Microsoft Visual
Studio\2022\Community\VC\Auxiliary\Build\vcvars64.bat"
SRC = "main.cpp"
OUTDIR = Path("logs")
OUTDIR.mkdir(exist_ok=True)
EXE = "program.exe"

# ======================
# PARAMETRY KOMPILACJI
# ======================
OPT = "/O2"      # można zmienić na /Od lub /Ox
RUNTIME = "/MD"  # można zmienić na /MT, /MDd, /MTd
BUILD_MODE = f'"O2_MD_x64"'

# ======================
# KOMENDA KOMPILACJI
# ======================
compile_cmd = f'"{VCVARS}" && cl {SRC} {OPT} {RUNTIME} /D BUILD_MODE={BUILD_MODE} /Fe{EXE}'

print("[INFO] Kompiluję:", compile_cmd)
start = time.perf_counter()
subprocess.call(compile_cmd, shell=True)
end = time.perf_counter()

print(f"[INFO] Czas kompilacji: {end - start:.3f}s")

# ======================
# URUCHOMIENIE PROGRAMU
# ======================
run = subprocess.run([EXE], capture_output=True, text=True)
log_name = OUTDIR / f"build_{datetime.datetime.now().strftime('%Y%m%d_%H%M%S')}.log"

with open(log_name, "w", encoding="utf-8") as f:
    f.write(run.stdout)
    f.write(f"\nCzas kompilacji: {end - start:.3f}s\n")

print(f"[OK] Wynik zapisano do: {log_name}")
```

Rozszerzenia dla jeszcze ambitniejszych

1. Dodaj pomiar czasu działania programu (`time.perf_counter()` przed i po `subprocess.run`).
2. Dopisz do logu rozmiar pliku `.exe` (`os.path.getsize(EXE)`).
3. Zrób automatyczne testy: sprawdź, czy wynik zawiera słowo *Hello*. Jeśli nie, wypisz błąd.
4. Dodaj możliwość wyboru trybu kompilacji z argumentów (`argparse`).
5. Zrób pętlę testującą różne konfiguracje `/O2`, `/Od`, `/Ox`, `/MD`, `/MT` i zapisz porównanie w CSV.

Continuous Integration (CI)

Ten skrypt to pierwszy krok w stronę CI.
W środowiskach takich jak **GitHub Actions**, **GitLab CI**, **Jenkins** czy **Azure DevOps**, proces:

1. Pobiera kod z repozytorium,
2. Automatycznie uruchamia kompilację,
3. Wykonuje testy,
4. Zapisuje logi i raporty.

To pozwala na natychmiastowe wykrywanie błędów po każdej zmianie w kodzie.

Zadania (do oddania w formie sprawozdania)

1. Uruchom skrypt w trybie `/O2 /MD` i `/Od /MDd`. Porównaj czasy kompilacji i wielkość plików `.exe`.
2. Dodaj do `main.cpp` licznik pętli `for` 10^6 iteracji i sprawdź, jak różni się czas działania między `/Od` a `/O2`.
3. Dodaj `#define FEATURE_X` i spraw, by program wypisywał dodatkową informację tylko wtedy, gdy jest aktywne.
4. Utwórz plik CSV `results.csv` z kolumnami: `mode, compile_time, exe_size, runtime`.
5. Dla chętnych: skonfiguruj prosty workflow GitHub Actions, który uruchomi `build_and_run.py` po każdym pushu.
