Warsztat Programisty – materiały do zajęć

Łukasz Kuszner, Radosław Ziemann

Uniwersytet Gdański
Wydział Matematyki, Fizyki i Informatyki
Instytut Informatyki
Zakład Optymalizacji Kombinatorycznej

13 stycznia 2025

8 Narzędzia do budowania aplikacji

W rozdziale 6 przyjrzeliśmy się jak można podzielić program na kilka plików i kompilować je oddzielnie. Przy okazji użyliśmy programu `make`, co uprościło wywołanie kompilatora.

W szerszym kontekście mówimy o procesie *budowania aplikacji* (ang. software build), w którym oprócz samego wytworzenia jego efektów (samodzielnych programów lub ich komponentów) interesuje nas zapewnienie:

- powtarzalności procesu,
- kontrola i zapewnienie jakości wytworzonych artefaktów,
- tworzenie różnych wersji w zależności od celu,
- efektywności procesu,
- łatwe dostarczenie produktu klientowi (ang. deployment).
- praca na kilku platformach sprzętowych i/lub systemowych (ang. Cross-platform software development)

Spośród narzędzi pomocnych w osiąganiu tych celów znamy już między innymi: edytor kodu, kompilator, debugger, system kontroli wersji i narzędzie automatyzujące kompilację (program `make`). Programista może używać każdego z tych narzędzi osobno lub jednego narzędzia łączącego je wszystkie w sobie, czyli *środowiska zintegrowanego* lub krócej IDE (ang. Integrated Development Environment).

8.1 Środowisko zintegrowane

Korzystanie ze środowiska zintegrowanego może się wydawać idealnym rozwiązaniem. zwłaszcza dla osoby, która już jakieś IDE poznała a teraz rzeczy, które przedtem wydawały się proste, stały się skomplikowane gdy próbujemy je robić używając każdego narzędzia osobno.

Jest to jednak ideał pozorny. Wśród problemów możemy wskazać następujące:

- Indywidualne preferencje członków zespołu mogą się istotnie różnić.
- Poszczególne środowiska IDE mogą nie być dostosowane do pracy z różnymi platformami (programowymi i sprzętowymi).
- Pliki konfiguracyjne projektu mogą nie być w pełni przenośna a ich wersjonowanie i śledzenie zmian nastręcza problemów.

8.2 Systemy budowania

Trudności napotkane przy tworzeniu oprogramowanie wieloplatformowego doprowadziły do powstania systemów, które na podstawie swoich plików konfiguracyjnych tworzą konfiguracje dla programów takich jak `make` lub IDE. Wśród takich systemów możemy wymienić `CMake`, `Meson`, czy `Basel` .

Typowy cykl pracy z projektem z punktu widzenia programisty wygląda następująco:

1. Pobierz z repozytorium wersję oprogramowania nad którą chcesz dalej pracować.
2. Skonfiguruj projekt na platformę na której chcesz pracować (sprzętową i programową) dla wybranego środowiska i/lub generatora.
3. Zbuduj projekt.
4. Sprawdź, czy działa.
5. Wykonaj potrzebne zmiany.

6. Sprawdź, czy działa.

7. Złóż wykonane zmiany do repozytorium projektu.

Dzięki systemowi budowania wszystkie powyższe punkty z wyjątkiem 5 i 7 powinny być wykonane w możliwie prosty, powtarzalny sposób.

Przykład

Przykładowo, dla repozytorium `git` skonfigurowanego w systemie `cmake` na komputerze klasy PC z systemem operacyjnym `Linux` mogłoby to wyglądać następująco:

```
git clone https://github.com/lkuszner/WP24.git
cd WP24 && mkdir build && cd build && cmake ../src/
make
 ./tests/test_power
 # wykonaj właściwą pracę nad projektem
 ./tests/test_power
 # Przygotuj złożenie (ang. commit), poddaj ocenie i wyślij do repozytorium
```

Polecenie `git clone` tworzy lokalną kopię zdalnego repozytorium. Kolejna komenda tworzy osobny katalog `build` na pliki będące wynikiem procesu budowania aplikacji i uruchamia `cmake` z domyślnymi parametrami. Wykonanie `make` buduje aplikację wraz z testami znajdującymi się w katalogu `tests`.

Zaletą systemu `cmake` jest fakt, że praca programisty w innym systemie operacyjnym z IDE (na przykład Visual Studio w systemie Windows albo XCode na Mac OS) może przebiegać analogicznie i zależeć od dokładnie tych samych plików zapisanych we wspólnym repozytorium.

8.3 CMake

System `CMake` to oprogramowanie z otwartym kodem źródłowym (ang. open-source sofware), które umożliwia programistom określanie parametrów kompilacji w prostym, przenośnym formacie pliku tekstowego. Ten plik jest następnie używany przez CMake do generowania plików projektu dla natywnych narzędzi kompilacji, w tym zintegrowanych środowisk programistycznych (IDE), takich jak Microsoft Visual Studio lub Apple Xcode, a także UNIX, Linux, NMake i Ninja.

Jeśli nad projektem pracuje wielu programistów na wielu platformach docelowych, to oprogramowanie będzie musiało zostać zbudowane na więcej niż jednym komputerze. Biorąc pod uwagę szeroki zakres zainstalowanego oprogramowania i opcji niestandardowych, które są związane z konfiguracją nowoczesnego komputera, istnieje wysokie prawdopodobieństwo, że dwa komputery z tym samym systemem operacyjnym będą się nieznacznie różnić. Wtedy `CMake` zapewnia wiele korzyści a w tym:

- Możliwość automatycznego wyszukiwania programów, bibliotek i plików nagłówkowych, które mogą być wymagane przez tworzone oprogramowanie. Obejmuje to możliwość uwzględnienia zmiennych środowiskowych i ustawień rejestru systemu Windows podczas wyszukiwania.

- Możliwość budowania w drzewie katalogów poza drzewem z kodami źródłowymi (ang. out-of-source). Umożliwia to programiście usunięcie całego katalogu kompilacji bez obawy o usunięcie plików źródłowych.

- Możliwość tworzenia złożonych, niestandardowych poleceń dla automatycznie generowanych plików źródłowych podczas procesu kompilacji, które są następnie kompilowane do oprogramowania.

- Możliwość wybierania opcjonalnych komponentów (na przykład wybierania bibliotek) w czasie konfiguracji, które mają zostać zbudowane.

- Możliwość łatwego przełączania się między kompilacjami statycznymi i dynamicznymi (ang shared).

- Obsługa równoległych kompilacji na większości platform.
- Możliwość testowania kolejności bajtów maszynowych i innych cech specyficznych dla sprzętu.
- Pojedynczy zestaw plików konfiguracji kompilacji, który działa na wszystkich platformach. Dzięki temu programiści nie muszą utrzymywać tych samych informacji w kilku różnych formatach w ramach projektu.

Przykład

Domyślna nazwa pliku konfiguracyjnego dla `CMake` to `CMakeLists.txt`. Minimalny taki plik określa wymaganą wersję `CMake`, nazwę projektu, plik wykonywalny i jego zależności, na przykład:

```
cmake_minimum_required(VERSION 3.10)
project(Hello)
add_executable(hello hello.c)
```

Mając powyższy tekst w pliku `CMakeLists.txt` oraz dowolny poprawny program w pliku `hello.c` możemy teraz przeprowadzić kompilację przy pomocy systemu `CMake`. Robimy to w innym katalogu niż przechowujemy te pliki. Przypuśćmy, że katalogiem przeznaczonym do pracy nad projektem jest `ProjectHello`, a w nim katalog z `src` zawiera pliki `CMakeLists.txt` oraz `hello.c`. Tworzymy osobny katalog do przeprowadzenia konfiguracji i budowania projektu (ang. out-of-source build):

```
mkdir build
cd build
cmake ../src/
cmake --build
```

Jeśli wszystko przebiegło pomyślnie, to możemy teraz uruchomić powstały program o nazwie `hello`. Te same polecenia w systemie `Windows` spowodują utworzenie pliku `hello.exe` zgodnie z konwencją nazywania plików w tym systemie.

8.4 Wersje Debug i Release

Pracując nad projektem programista typowo używa innych opcji niż docelowa wersja oprogramowania przeznaczona dla klienta. W wersji roboczej (ang. Debug) ustawia się opcje dołączające informacje dla debuggera i wyłącza optymalizacje kodu. W wersji przeznaczonej dla klienta przeciwnie: wyłączamy informacje dla debuggera i włączamy optymalizacje kodu.

Korzystając z `CMake` mamy gotowe domyślne zestawy opcji dla różnych wersji. W szczególności możemy ustawić wersję `release` ustawiając opcję: `-DCMAKE_BUILD_TYPE=Release`. Pozostałe komendy pozostają te same:

```
mkdir release_build
cd release_build
cmake -DCMAKE_BUILD_TYPE=Release ../src/
cmake --build
```

Ćwiczenie 37. Przeprowadź kompilację prostego projektu przy użyciu narzędzia `CMake`. Stwórz przynajmniej dwie wersje: `Debug` i `Release` w osobnych katalogach. W trakcie budowania obu wersji włącz wypisywanie pełnych poleceń, zamiast: `cmake --build` użyj `make VERBOSE=1`. Porównaj opcje kompilatora i rozmiar powstałych plików wykonywalnych (powinny się różnić).