Лекція 2. Знайомство з програмуванням на C++.

Версія 25 лютого 2023 р.

Анотація

Ця лекція знайомить з процесом розробки та компіляції програм на мові C++, використанням (мета-)системи побудови програм CMake.

Зміст

1 Як побудована найпростіша програма в C++?

Кожен проект C++ має мати одну єдину функцію `main()`, що може приймати вихідні аргументи, або ні, яка буде викликана при запуску скомпільованого файлу. Вона має наступну форму:

```
int main() {
  /* your code here */
  return 0;
}
```

За конвенцією вона має повертати ціле число, де 0 означає успішне завершення програми, а будь-яке інше – конкретну чи загальну помилку чи неочікувану поведінку при виконанні.

Будь-яка програма C++ використовує так звані #include директиву пре-процесора. Вона дозволяє нам підключати додатковий функціонал у вигляди бібліотек, як стандартних, так і сторонніх, або написаних нами. Вони мають наступний формат:

- `#include <file>` – для системних файлів,
- `#include "file"` – для локальних файлів.

Майте на увазі, що різниця в формату не строга, але її варто дотримуватись.

Додаються ці команди на початку вашого .h (.hpp), або .cpp (.cc) файлів, і, перед початком компіляції, препроцесор замість цих директив вставить повністю вміст відповідних файлів. Існують також і інші директиви, з повним списком можна ознайомитись тут, але їх (часте) використання не заохочується.[1]

2 Процес компіляції

C++ – це компільована мова програмування, що означає, що спеціальна програма (компілятор) мусить спочатку перетворити її вихідний код у файл з машинним кодом, який користувач вже може запустити на своєму комп'ютері.

Для того, щоб скомпілювати вашу першу програму, достатньо виконати наступну просту команду [2]:

```
> c++ oneFileProgram.cpp
```

Clang побудує виконуваний файл, який буде знаходитись в поточній теці, та буде називатись `a.out`. Передати свою назву модна за допомогою прапорця `-o`, разом з іншими налаштуваннями, наприклад, стандарт C++, який ми плануємо використовувати для нашого проекту:[3]

[1] Ще одним винятком є використання `#pragma once` для уникнення включення того самого header файлу кілька разів, або використання `#ifndef` для досягнення тієї самої мети, що є старішим підходом.

[2] В цьому курсі подібні приклади будуть даватись в вигляді команди терміналу з використанням компілятора Clang (визирається за допомогою `c++`, або `clang++`; для GCC – `g++`, або `gcc`).

[3] Компілятори мають силу силенну подібних налаштувань та опцій, які дуже важливі для реальних проектів. Знайти їх можна в документації до вашого компілятора.

```
> c++ -std=c++17 -o my_program oneFileProgram.cpp
> ./my_program 10
Is 10 positive? Answer: yes.
> ./my_program -10
Is -10 positive? Answer: no.
```

Зауважте, програма `oneFileProgram.cpp` має приклади роботи з потоками `iostream`, а також `stringstream`. Важливо також знати файлові потоки `fstream`. Вони мають величезну важливість в роботі (а також і в завданнях). На щастя, вони мають спільний інтерфейс, тому робота з одним, дуже сильно нагадує роботу з іншими.

Чи завжди процес компіляції буде таким простим? Звичайно ні. Можна сказати, він ніколи в реальному житті такий простий. Уявімо, для зручності і розширюваності нашої програми з попереднього прикладу, ми розіб'ємо її на декілька файлів (див. `multiFileProgram.cpp`). Якщо ми тепер виконаємо ту саму команду, отримаємо помилку

```
> c++ -std=c++17 -o my_program multiFileProgram.cpp
Undefined symbols for architecture x86_64:
  "isPositive(int)", referenced from:
      _main in multiFileProgram-528f77.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use
    -v to see invocation)
```

Звичайно, для такого простого випадку, є швидке вирішення – просто додати всі файли, де знаходиться наш вихідний код до команди, щоб компілятор знав де шукати наш код:

```
> c++ -std=c++17 -o my_program multiFileProgram.cpp
    funcs.cpp
```

Але, це не завжди є бажаним, якщо ми хочемо працювати з нашими `funcs` файлами, як з бібліотеками, або ж підключити реальні сторонні бібліотеки. Але спочатку, давайте зрозуміємо суть помилки, яку ми побачили.

Процес компіляції складається з чотирьох кроків:

1. пре-процес (попередня обробка),
2. компіляція (яка дає назву всьому процесу),
3. збірка,
4. лінковка (зв'язування).

Розберемо трохи детальніше кожний з цих кроків:

1. **Пре-процес**. Викликається пре-процесор, який робить попередню обробку вихідного коду, згідно директив, що зазначив програміст: вставляє код з header файлів, видаляє та залишає окремі секції коду, видаляє коментарі. Щоб запустити цей процес окремо в Clang і подивитись на результат, треба виконати команду `clang++ -E main.cpp > main.i`.
2. **Компіляція**. Компілятор перетворює C++ код на команди асемблера. Цей крок запускається командою `clang++ -S main.i`, на виході отримуємо файл `main.s`.
3. **Збірка**. Код асемблера перетворюється на машинний двійковий код. Цей крок запускається командою `clang++ -c main.s`, на виході отримуємо файл `main.o`.
4. **Лінковка**. Весь машинний код з .o файлів збирається в виконуваний файл. Команда `clang++ main.o -o main`.

Повертаючись до нашої помилки компіляції, бачимо, що вона завершилась на етапі лінкування, тому що лінкер не зміг знайти код функції `isPositive`, що логічно, тому що ми не додали файл з ним до проекту. Подивимось як додати наш файл з допоміжною функцією як бібліотеку. Бібліотека – це збірка пов'язаних сутностей. Вони бувають статичними, тобто такими, які є вбудованими в ваш кінцевий файл, та динамічними, які знаходяться в системі, на якій запускається програма. Наш файл ми будемо використовувати як статичну бібліотеку, у той час ви бачили, що такі бібліотеки як `iostream` в нашому прикладі були динамічними. Щоб створити свю статично бібліотеку (у вигляді архіву), можна використати наступну команду

```
> ar rcs libfuncs.a funcs.o
```

Зауважте, що файл `funcs.cpp` має бути вже скомпільований у `funcs.o`. Щоб Clang знайшов її, треба передати шлях до її розташування за допомогою `-L` а також її ім'я за допомогою `-l funcs`. Отже повний процес компіляції буде виглядати так:

```
> c++ -std=c++17 -c funcs.cpp -o funcs.o
> ar rcs libfuncs.a funcs.o
> c++ -std=c++17 multiFileProgram.cpp -L . -l funcs -o
   my_program
```

3 Системи побудови коду

Як ви вже бачили, для більших програм, ніж ті, що складаються з однієї функції `main()`, процес компіляції коду (побудови програми) стає неінтуїтивним та важким для людини. На додачу, не просто може бути пояснити іншій людині як їй побудувати вашу програму в себе на комп'ютері. На порятунок приходять так звані системи побудови коду. Історично, вони існували як shell скріпти, що виконують прописані заздалегідь кроки. Ця практика пізніше переросла в системи побудови на кшталт Unix MakeFiles. Але все ще користуватись ними для великих проектів було важко, тому що треба було підтримувати багато супутніх make файлів. Щоб зарадити цьому, з'явилися мета-білд системи, з яких для C++ найбільш поширеною є CMake.

За допомогою CMake, наш процес компіляції можна задати наступним чином

```
# instead 'c++ -std=c++17 -c funcs.cpp -o funcs.o && ar rcs
    libfuncs.a funcs.o'
add_library(funcs funcs.cpp)
# instead 'c++ multiFileProgram.cpp -o my_program'
add_executable(my_program multiFileProgram.cpp)
# instead '-L . -l funcs'
target_link_libraries(my_program funcs)
```

Погодьтесь, цей скрипт є набагато зрозумілішим і легшим в редагуванні. Повний приклад можна подивитись в файлі `CMakeLists.txt`, що додається до цієї лекції. `CMakeLists.txt` – це вимога CMake, назва за замовченням, щоб він знав, де ви зберігаєте його інструкції, що полегшує життя, дозволяючи не писати назву файлу при кожному виклику. Але ми ще не бачили, як саме робити цей виклик! Порядок роботи наступний:

1. Створити окрему директорію, де будуть зберігатись всі згенеровані, тимчасові та допоміжні файли, а також і сама вихідна програма (за замовченням, але її можна інсталювати будь де): `mkdir build`.
2. Перейти туди: `cd build`.
3. Запустити CMake, вказавши розташування корінного файлу `CMakeLists.txt`: `cmake ...` За замовченням, на Unix-подібних платформах використовується стистема MakeFiles за замовченням. Вказати іншу можна за допомогою параметра `-G`. Повний список генераторів, доступних на окремо взятому комп'ютері, можна подивитись за допомогою `cmake --help`.
4. Запустити саму систему побудови, для якої CMake згенерував все необхідне: `make`, або більш універсально: `cmake --build .`, щоб дати

CMake самому викликати потрібний інструмент.

В кінці, ви побачити вашу програму в `./build/my_program`. Всі згенеровані файлі зберігаються в цій теці, яку можна завжди безболісно видалити, якщо щось пішло не так, і створити заново.

Деякі основні корисні команди CMake:

- `cmake_minimum_required` – задає мінімальну версію CMake, що працює з цим проектом. Мусить бути першою командою будь-якого `CMakeLists.txt`.
- `project` – назва вашого проекту.
- `set` – дозволяє задавати та створювати змінні.
- `find_library` – знаходить скомпільовану бібліотеку, наприклад

```
find_library(TOOLS
        NAMES tools
        PATHS ${LIBRARY_OUTPUT_PATH})
# Library info is in TOOLS variable
target_link_libraries(<some_binary> ${TOOLS})
```

- `find_path` – додає шлях з header файлами до проекту, наприклад

```
find_path(SOME_PKG_INCLUDE_DIR include/some_file.hpp
    <path1 > <path2 > ...)
include_directories(${SOME_PKG_INCLUDE_DIR})
```

- `find_package` – автоматизує ці кроки, працює одразу з популярними бібліотеками, такими як OpenCV.

Завдання

Знайдіть якийсь простий проект (ваш або чужий), або напишіть свій, як в прикладах цієї лекції і додайте до неї робочий `CMakeLists.txt` файл. Подивиться в документації який додатковий функціонал CMake може стати вам в нагоді. Приклади програм можна подивиться в GitHub репозиторії онлайн курсу The Complete C++ Developer Course.

Корисні посилання

- C++ Reference
- C++ Tutorial
- CMake Documentation

- GCC Manual
- Clang Manual