به نام خدا

مستندِ آشنایی با **Cmake**

نیم‌سال اول ۹۷-۹۶

دانشکدهٔ مهندسی کامپیوتر
دانشگاه صنعتی شریف

نویسندگان **محمد شهیدی، محمد امین سالارکیا**
ویراستاری ادبی **سینا ریسمانچیان**

- این مستند جهت آشناییِ اولیه‌ی شما با Cmake آماده شده است.
- در ادامه‌ی پروژه به cmake نیاز پیدا می‌کنید.
- هرگونه سوال در موردِ Cmake را می‌توانید در کوئرا بپرسید.

مقدمه

در صنعت توسعه برنامه‌سازی استفاده از برنامه‌های پیش ساخته در اولویت قرار دارد. به این معنی که اگر لازم شد الگوریتم پیاده‌سازی شود، بهتر است از نسخه‌های پیاده‌سازی‌شده در خود زبان یا Libraryهای دیگر که به برنامه اضافه می‌شوند استفاده شود چرا که صحت آنها تایید شده است و مهم‌تر از آن در زمان صرفه‌جویی میشود. اما همانطور که احتمالا متوجه آن شده‌اید بسیاری از توابع پرکاربرد و نسبتا ابتدایی در زبان C به صورت آماده موجود نیست و باید Library آنها از منابع دیگر ذخیره و سپس استفاده شود. به عنوان مثال Libraryهایی برای کار با گرافیک به طور پیشرفته یا برای اتصال برنامه به شبکه. معمولا این Libraryها به صورت فایل‌های Header و تعدادی فایل با فرمت‌های اجرایی (مثلا dll.، a. و so.) هستند. البته اگر Library مورد نظر Open-Source باشد فایل‌های اصلی پیاده‌سازی در کنار فایل‌های Header و اجرایی وجود دارند که به فرمت C یا Cpp هستند و معمولا در پوشه ای به نام Source Files قرار دارند. در ادامه چگونگی اتصال Library ها به برنامه توسط Cmake توضیح داده خواهد شد.
احتمالا تا کنون از دستور زیر در Command-Line سیستم عامل خود استفاده کرده‌اید :

```
> gcc main.c -o ApplicationName.out
```

یا این دستور که کد را Optimize میکند:

```
> gcc main.c -O2
```

به قسمت آخر دستورهای بالا Parameterهای دستور گفته می‌شود که در واقع چگونگی اجرای دستور را به عامل دستور نشان می‌دهد. با یک جستجوی ساده متوجه می‌شوید که برای دستور کامپایل با کامپایلر gcc تعداد زیادی Parameter وجود دارد که گاهی برای یک پروژه تعدادی از آنها همزمان استفاده می‌شوند. به همین ترتیب ممکن است در طول پروژه تعدادی از این پارامترها به دستور اضافه شوند. استفاده مداوم از دستور کامپایل با طول بلند دشوار است و امکان اشتباه وجود دارد. مهم‌تر از آن در تیمی که روی یک پروژه کار می‌کنند همه باید یک دستور را برای کامپایل پروژه استفاده کنند و این دستور باید بین اعضا به اشتراک گذاشته شود. این قضیه نیز با استفاده از cmake به راحتی حل می‌شود.
شاید تا الان بیشتر برنامه‌های شما فقط روی یک فایل بوده‌اند اما پروژه‌های بزرگ معمولا از تعداد زیادی فایل تشکیل شده‌اند که همه با هم یک برنامه را تشکیل می‌دهد. این فایل به تنهایی کامپایل نمی‌شوند و برای ساختن برنامه باید دستوری شامل همه آنها داشته باشید یا قسمتی از آنها را به صورت Library هایی برای بقیه فایل ها در نظر بگیرید. این مورد مانند مورد قبل اجرای دستور کامپایل را سخت می‌کند و باعث اشتباه می‌شود.

Cmake

Cmake برنامه‌ای مستقل از سیستم عامل و Open-Source است که برای مدیریت فرایند ساخت برنامه مورد استفاده قرار می‌گیرد. پروژه‌ها با استفاده از لیست فایل‌های Cmake روند ساخت خود را مشخص می‌کنند. این لیست فایل‌ها در directory های مجاز با نام CMakeLists.txt قرار دارند. پس از کامل شدن لیست فایل ها عملیات ساخت پروژه توسط برنامه Cmake انجام میشود. نصب این برنامه به راحتی انجام می‌شود اما برای اطمینان بیشتر در بخش زیر نکاتی در این زمینه وجود دارد.

نصب Cmake

برای استفاده از Cmake پیشنهاد می‌شود که از CLion IDE استفاده شود. این IDE به همراه خود Cmake دارد و با ایجاد پروژه لیست فایل‌های Cmake را ایجاد می‌کند و با اجرای برنامه از طریق Cmake فرایند ساخت انجام می‌شود. اما اگر می‌خواهید از طریق Terminal از lrCmake[استفاده کنید به طریق زیر می‌توانید آن را نصب کنید.

- Windows : برای سیستم 32 bit این فایل و برای سیستم 64 bit این فایل را دانلود نمایید.
توجه کنید حین نصب گزینه ی ... Do not add Cmake to را انتخاب نکنید و یکی از دو گزینه پایین لیست را انتخاب نمایید. پس از نصب برای تست کردن ابتدا سیستم را restart کرده و سپس CMD را باز کنید و دستور زیر را وارد نمایید :

```
> cmake --version
```

- Linux : کافیست ترمینال را باز کنید و دستور زیر را در آن وارد کنید.

```
> sudo apt install cmake
```

- Mac : اگر برنامه brew در سیستم شما نصب نشده است با وارد کردن کد زیر در Terminal می توانید آن را نصب کنید :

```
> mkdir homebrew && curl -L https://github.com/Homebrew/brew/tarball/master | tar xz
    -strip 1 -C homebrew
```

سپس با استفاده از برنامه brew با وارد کردن کد زیر میتوانید Cmake را نصب کنید.

```
> brew install cmake
```

Cmake lists

همانطور که گفته شد cmake اطلاعات مورد نیاز را از لیست فایل ها دریافت میکند.
در لیست فایل ها میتوان متغیر تعریف کرد یا از متغیرهای پیش تعریف شده توسط cmake استفاده کرد. معمولا در cmake از متغیر ها برای نگه داشتن آدرس ها یا لیستی از فایل ها یا پارامتر های مختلف برای کامپایل استفاده می‌شود. نحوه‌ی تعریفِ متغیر در cmake به صورتِ زیر است:

```
set( MY_PATH "src/main")
```

و نحوه استفاده از متغیرها در cmake به صورتِ زیر است:

```
${MY_PATH}
```

در کدِ بالا به جای MY_PATH مقدارِ src/main قرار می‌گیرد.
در ادامه برای سادگی کار یک لیست فایل نشان داده شده تا با دستورات آن آشنا شوید.

```
cmake_minimum_required(VERSION 2.8)

project(hello)

set(CMAKE_BINARY_DIR ${CMAKE_SOURCE_DIR}/bin)
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR})
set(LIBRARY_OUTPUT_PATH ${CMAKE_BINARY_DIR})

include_directories("${PROJECT_SOURCE_DIR}")

SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
```

```
12
13 SET(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)
14
15 add_executable(hello ${PROJECT_SOURCE_DIR}/test.cpp)
```

در خط اول در کد بالا کمینه ورژن مناسب برای ساخت پروژه مشخص شده است. اگر ورژن Cmake از آن مقدار کمتر باشد فرایند ساخت متوقف می‌شود. در خط سوم نام پروژه مشخص شده است. Cmake تعدادی متغیر Global دارد که می توانیم از آنها در لیست فایل‌ها استفاده کنیم و آنها را تغییر دهیم.
متغیر CMAKE_SOURCE_DIR ، در واقع همان directory است که توسط command-line به دستور Cmake داده شده است.
متغیر EXECUTABLE_OUTPUT_PATH، این متغیر جایی که خروجی‌های اجرایی برنامه در آن ریخته خواهند شد را نشان میدهد. توجه کنید که این متغیر باید توسط دستور set تنظیم شود.
متغیر LIBRARY_OUTPUT_PATH ، اگر برنامه خروجی library داشته باشد این متغیر جای آنها را مشخص میکند.
متغیر PROJECT_BINARY_DIR ، directory که در آنجا دستور cmake اجرا شده است را نگه می‌دارد.
برنامه Cmake ، directoryهایی دارد که اگر قرار باشد فایلی را پیدا کند آن مکان ها را جستجو میکند. به طور پیشفرض مکان هایی وجود دارند اما برای اضافه کردن directory باید از دستوری مانند دستور خط ۹ استفاده کرد.
دستور خط ۱۱ متغیری که قبل تر در خط ۶ مقدار دهی شده بود را دوباره مقدار می دهد. همواره آخرین تغییر اعمال می‌شود و بنابراین نبودن خط ۶ تفاوتی ایجاد نمی‌کند.
در نهایت دستور خط ۱۵ فایلی را برای کامپایل به Cmake معرفی می‌کند. همچنین بیان میکند که خروجی را hello نام‌گذاری کند. البته این که این فایل کجا ذخیره شود قبل‌تر مشخص شده بود.
توجه کنید که در نوشتن کلمات دستوری لیست فایل ها بزرگی یا کوچکی حروف مهم نیست. اگر ابتدای خط با tab شروع بشود به معنی اسکوپ جدید است و اگر بی مورد باشد منجر به ارور می‌شود.
لینک مثال بالا

در ادامه یک مثال کامل‌تر را بررسی می‌کنیم :

```
1  cmake_minimum_required(VERSION 2.8.12)
2
3  project(glui)
4  set(PROJECT_VERSION 2.37)
5
6  find_package(GLUT REQUIRED)
7  find_package(OpenGL REQUIRED)
8
9  if (CMAKE_CXX_COMPILER_ID STREQUAL "Clang")
10     set (CMAKE_CXX_FLAGS "--std=c++11 ${CMAKE_CXX_FLAGS}")
11 elseif (CMAKE_CXX_COMPILER_ID STREQUAL "GNU")
12     set (CMAKE_CXX_FLAGS "--std=c++11 ${CMAKE_CXX_FLAGS}")
13 endif ()
14 SET(SOURCE_FILES
15      tools/ppm.c
16      tools/ppm2array.c
17 )
18 add_executable(ppm2array ${SOURCE_FILES})
```

در خط چهارم این فایل یک متغیر جدید تعریف شده است و می‌توان از آن استفاده کرد. خط ششم و هفتم چک می‌کند که آیا libraryهای مربوطه بر روی سیستم نصب شده‌اند یا نه. اگر نصب نباشند فرایند ساخت متوقف میشود(کلید واژه REQUIRED). در ادامه یک مثال از عبارت شرطی if را مشاهده می‌کنید. کلمه STREQUAL مانند "==" در زبان C عمل می‌کند. خط ۱۰ و ۱۲ یک پارامتر به پارامتر های دستور کامپایل اضافه می‌کند. مانند پارامتر O2- که در مقدمه درباره آن صحبت شد. در خطوط ۱۴ تا ۱۷ نیز متغیر دیگری تعریف شده است که نام فایل هایی که قرار است کامپایل شوند را مشخص می‌کند. زمان کامپایل، Cmake به دنبال این فایل‌ها در directoryهایی که در لیست‌ها مشخص شده است می‌گردد و در صورت پیداکردن کامپایل می‌کند. خط ۱۸ نیز دو فایل C را با هم کامپایل می‌کند تا فایل اجرایی ppm2array ساخته بشود. این دو فایل به هم مربوط هستند و اگر جدا کامپایل می‌شدند ممکن بود ارور کامپایل بگیرند.

External library

برای استفاده از کدهای از پیش ساخته شده معمولا یک خروجی به صورتِ فایل‌های کتابخانه‌ای از آن‌ها می‌گیرند.در این خروجی سورسِ کدِ زده شده برای ساختن این کتابخانه‌ها قابل مشاهده نیست و این فایل ها فقط به صورتِ binary توسط سیستم عامل قابل اجرا هستند. از این فایل‌ها در

کدهای مختلف می‌توان استفاده کرد.
دو نوع پرکابرد کتاب‌خانه‌ها static و shared هستند.

Shared

پسوندِ این نوع کتابخانه‌ها so. (یا در ویندوز dll. و در مک dylib.) است. کدهایی که ازین کتابخانه‌ها استفاده میکنند در حینِ اجرا به آن‌ها رجوع میکنند. بنابراین این فایل‌ها در حینِ اجرا باید در کنار فایل اجراییِ نهایی(مثلا exe) باشند.

Static

پسوند این نوع کتابخانه‌ها a. (یا در ویندوز lib.) است. کدهایی که ازین کتابخانه‌ها استفاده میکنند در زمانِ کامپایل به آن‌ها رجوع میکنند. بنابراین نیازی نیست که این فایل ها در حین اجرا کنارِ فایلِ اجراییِ نهایی باشند.

حال قبل از اینکه سراغِ لینک کردنِ یک کتابخانه برویم کمی دربارهٔ هدر فایل ها صحبت میکنیم.

Header files

کد زیر را در نظر بگیرید:

main.c:

```c
int sum(int a, int b){
    return (a+b);
}

int main(){
    printf("%d", sum(2,4));
}
```

این کد را میتوان به صورتِ زیر هم نوشت:

main.c:

```c

int sum(int a, int b);

int main(){
    printf("%d", sum(2,4));
}

int sum(int a, int b){
    return (a+b);
}
```

در واقع در بالای فایل میتوان پروتوتایپِ توابع و متغیر های موردِ استفاده را نوشت. اما راهِ بهتری که معمولا از آن استفاده میشود این است که کنارِ فایلِ main.c یک فایلِ main.h هم قرار داد به این صورت:

main.c:

```c
#include "main.h"
int main(){
    printf("%d", sum(2,4));
}

int sum(int a, int b){
    return (a+b);
}
```

main.h:

```c
int sum(int a, int b);
```

به اینگونه فایل های (با فرمتِ h.) مانند main.hکه پروتوتایپِ توابع و متغیرهایی که قرار است پیاده‌سازی شوند را معرفی میکنند فایل‌های هدر میگویند.
برای لینک کردنِ کتابخانه ها به پروژه علاوه بر فایلِ binary کتابخانه باید فایل های header مربوط به آن کتابخانه هم کنارِ پروژه باشند. حال به بررسیِ یک مثال برای استفاده از یک static library میپردازیم.

```
cmake_minimum_required(VERSION 3.8)
project(cmake_link_library)

add_library(CMAKE_EXERCISE STATIC IMPORTED)
set_target_properties(CMAKE_EXERCISE PROPERTIES IMPORTED_LOCATION "../libmylib.a")
set(SOURCE_FILES main.c)
add_executable(cmake_link_library ${SOURCE_FILES})
target_link_libraries(cmake_link_library CMAKE_EXERCISE)
```

در این پروژه از یک کتابخانه‌ی خارجی که دارای یک تابع hello است که کارِ آن تابع، چاپِ hello world هست در کدِ main.c استفاده میکنیم.لینکِ گیت‌هابِ این پروژه در ادامه آمده است. میتوانید از طریقِ clion این پروژه را باز کرده و اجرا کنید:

https://github.com/M-amin-s/cmake_link_library

در خطِ پنجم به cmake وجودِ کتابخانه‌ای به اسمِ CMAKE_EXERCISE را که static است و به پروژه import میشود را به cmake میفهمانیم. در خطِ ششم آدرسِ فایل مربوط به کتاب‌خانه را ست میکنیم. دقت کنید چون تمامِ اتفاقات در فولدرِ cmake-build-debug اتفاق میفتند برای آدرس دادن از .. استفاده کرده‌ایم. در خطِ هشتم هم این library را به پروژه‌ی اجرایی لینک میکنیم.(با دیدنِ این پروژه در گیت‌هاب احتمالا ابهامات احتمالیتان برطرف میشود)
پس از کامل شدن لیست ها برای ساختن برنامه باید درون command-line به جایی که پروژه وجود دارد(یعنی فایلِ cmakelists.txt)و دستور زیر را وارد نمایید:

```
cmake .
make
```

بعد از اینکار فایلِ باینریِ تولید شده در فولدرِ cmake-build-debug قرار میگیرد.
دقت کنید این مستند جهتِ آشناییِ اولیه‌ی شما با cmake بود.طبیعتا برای یادگیری و تسلطِ کامل به سرچ و تمرین نیاز دارید.

تمام حجم قفس را شناختیم، بس است...
بیا به تجربه در آسمان پری بزنیم
قیصر امین‌پور