CMake チートシート – CMake の概要

このチートシートは, CMake がどのように機能し, ソフトウェアプロジェクトを構成するためにどう使用するのかを説明します。
ドキュメントと CMake の例は, `https://github.com/mortennobel/CMake-Cheatsheet` から入手できます[*1]。

■CMake-単純な C ++ プロジェクトの作成

CMake は, クロスプラットフォームのソースコードプロジェクトを所与のプラットフォームに対して構築する方法を設定するためのツールです。
小さなプロジェクトは次のように構成されるでしょう。

```
CMakeLists.txt
src/main.cpp
src/foo.cpp
src/foo.hpp
```

このプロジェクトには, src ディレクトリにある 2 つのソースファイルと同じディレクトリの include ディレクトリ中にある 1 つのヘッダーが含まれています。
このプロジェクトで CMake を実行ためには, バイナリディレクトリが必要です。本ディレクトリ (バイナリディレクトリ) は, プロジェクトの構築に関連する全てのファイルが含まれるため, バイナリディレクトリを新規に作成することをお勧めします。
問題が発生した場合は, バイナリディレクトリを削除し, 最初からやり直すことが可能です。
CMake を実行しても最終的な実行可能ファイルは作成されません, その代わりに, Visual Studio, XCode, または makefile のプロジェクトファイルを生成します。プロジェクトを構築には, これらのツール (Visual Studio, XCode, または, make) を使用します。

■CMakeLists.txt を理解する

CMake を用いてプロジェクトファイルを生成するために, `CMakeLists.txt` が必要となります。`CMakeLists.txt` には, プロジェクトの構成及びプロジェクトをどう構築するのかを記載します。
例 1 の場合, CMakeLists.txt ファイルは以下のようになります:

```
cmake_minimum_required (VERSION 2.9)

# プロジェクト名の設定
project (HelloProject)

# main.とcppfoo.をコンパイルcpp, リンクし,
# という実行形式を生成Hello
add_executable(Hello src/main.cpp src/foo.cpp)
```

まず, 使用する CMake の最小バージョンを定義します。次に, プロジェクト名をコマンド `project()` を使用して定義します。
プロジェクトには複数のターゲット (実行可能ファイルまたはライブラリ) を含めることができます。
このプロジェクトは `main.cpp` and `foo.cpp` の 2 つのソースファイルをコンパイル, リンクすることで構築される `Hello` という単一の実行可能ターゲットを定義します。
2 つのソースファイルをコンパイルする際に, コンパイラはヘッダーファイル `foo.h` を検索します。
両方のソースファイルが`#include "foo.hpp"`を使用しているため, 両方のソースファイルは, `foo.h` に依存しています。
この場合, ファイルはソースファイルと同じ場所にあるため, コンパイラーがヘッダーファイルの検索時に問題を引き起こすことはありません。

■CMake スクリプト言語

CMakeLists.txt は, コマンドベースのプログラミング言語によって, 構築手順を記載します。 CMakeLists.txt 中のコマンドは文字の大小を区別せず同様に扱われます。また,

[*1] 日本語版は, `https://github.com/takeharukato/CMake-Cheatsheet` から入手できます。

CMakeLists.txt のコマンドに引数列を与えることが可能です。

```
# コメント
COMMAND( 引数 )
ANOTHER_COMMAND() # このコマンドは引数をとらない
YET_ANOTHER_COMMAND( 複数行に渡って記載される
                # もう一つのコメント引数
    )
```

CMake スクリプトにも変数があります。変数は, CMake, または, CMake スクリプト中で定義することが可能です。**set(パラメタ 値)** コマンドは, 与えられたパラメタに値をセットします。**message(値)** コマンドは, コンソールに値を表示します。変数に設定されている値を取得するためには, **${変数名}**を使用し, 変数名を値に置き換えます。

```
cmake_minimum_required (VERSION 2.9)

SET( x 3 )       # x = "3"
SET( y 1 )       # y = "1"
MESSAGE( x y ) # "xyを表示"
MESSAGE( ${x}${y} ) # を表示"31"
```

すべての変数値は, テキスト文字列です。 テキスト文字列は, (**IF()** や **WHILE()** 内で使用された場合,) ブール評価式として評価されます。”FALSE”, “OFF”, “NO”, または, “-NOTFOUND” で終わる文字列の評価値は, 偽になります。それ以外は, 真になります。セミコロンで区切ってリストを指定することで, 複数の値を表すことができます。

```
cmake_minimum_required (VERSION 2.9)

SET( x 3 2) # x = "3;2"
SET( y hello world !) # y = "hello;world;!"
SET( z "hello world!" ) # y = "hello world!"
MESSAGE( ${x} ) # "xyを表示"
MESSAGE( "y = ${y} z = ${z}" )
# y = hello;world;! z = hello world! を表示
```

FOREACH (変数 値) コマンドを用いることでリストを反復することができます:

```
cmake_minimum_required (VERSION 2.9)

SET( x 3 2) # x = "3;2"
FOREACH (val ${x})
  MESSAGE(${val} )
ENDFOREACH(val)

# 表示:
# 3
# 2
```

■コンパイルオプションの公開

(CMake を実行する) エンドユーザーは, 変数の値を変更することができます。通常この機能は, ファイルの場所, マシンアーキテクチャ, 文字列値など, 構築に関する定義済みプロパティを変更するための変数の値を変更するために使用されます。
set(<変数> <値> CACHE <型> <文書文字列>) コマンドは, 変数をデフォルト値に設定しますが, 構築時に cmake ユーザーが設定値を変更できるようにします。型は次のいずれかである必要があります:

- FILEPATH = ファイル選択ダイアログ。
- PATH = ディレクトリ選択ダイアログ。
- STRING = 任意の文字列。
- BOOL = ブール値を表す ON/OFF チェックボックス。
- INTERNAL = GUI エントリなし (永続変数として使用)。

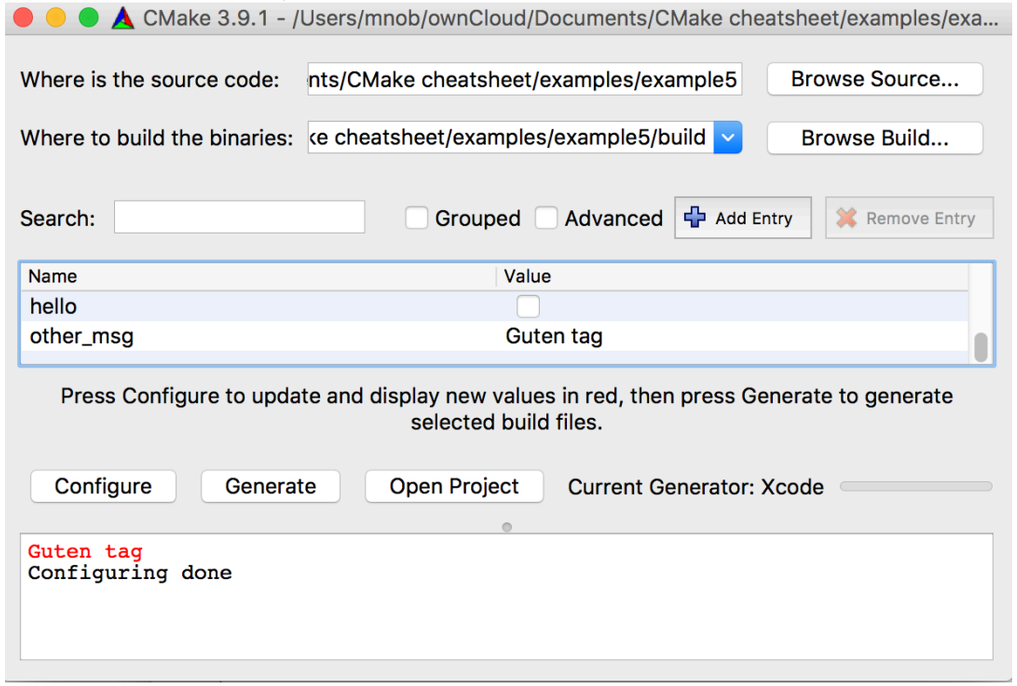
以下の例では, “Hello” または, 設定用の変数 hello や other_msg の内容に基づく代替文字列を表示するようにユーザが設定できるようにします。

```
cmake_minimum_required (VERSION 2.9)

SET(hello true CACHE BOOL "If true write hello")
SET(other_msg "Hi" CACHE STRING "Not hello value")
IF (${hello})
        MESSAGE("Hello")
```

```
ELSE (${hello})
        MESSAGE(${other_msg})
ENDIF (${hello})
```

プロジェクトの設定時に, CMake は公開オプションをユーザに問い合わせます。



CMake ユーザが入力した値は, テキストファイル `CMakeCache.txt` にキーと値の組 (*key-value pair*) として保存されます:

```
//  ....
//を表示hello
hello:BOOL=OFF

//以外の値hello
other_msg:STRING=Guten tag
//  ....
```

■複雑なプロジェクト

複数の実行形式と複数のライブラリを含むプロジェクトがあります。例えば, ユニットテストとプログラムの双方を持つような場合です。この場合, 各サブプロジェクトをサブフォルダに分割するのが普通です。例えば:

```
CMakeLists.txt
somelib/CMakeLists.txt
somelib/foo.hpp
somelib/foo.cpp
someexe/CMakeLists.txt
someexe/main.cpp
```

メイン `CMakeLists.txt` は, プロジェクトの基本的な設定を含みつつ, サブプロジェクトをインクルードします:

```
# CMakeLists.txt
cmake_minimum_required (VERSION 2.9)

# 名の設定project
project (HelloProject)

add_subdirectory(somelib)
add_subdirectory(someexe)
```

最初のライブラリ Foo は, `somelib` ディレクトリ中のソースからコンパイルされます:

```
# somelib/CMakeLists.txt

# foo.をコンパイルcpp, リンクする
add_library(Foo STATIC foo.cpp)
```

最終的に, 実行形式 Hello は, Foo ライブラリをリンクするようにコンパイルされます。ターゲット名は, 実際のパスではなく, ここで使われることに注意してください。main.cpp は, ヘッダファイル Foo.hpp を参照するため, somelib ディレクトリをヘッダサーチパスに追加しています。

```
# someexe/CMakeLists.txt

# をヘッダサーチパスに追加するsomlib
```

```
include_directories(../somelib/)

add_executable(Hello main.cpp)

# Foo ライブラリをリンクする
target_link_libraries(Hello Foo)
```

■ソースファイルの検索

所定のディレクトリ内のファイル群を一つ以上の検索パターンに従って自動的に検索するためには `find(GLOB 変数名 パターン)` を使用します。以下の例でソースファイルとヘッダファイルの双方がプロジェクトに追加されることに留意ください。これは, プロジェクトをコンパイルするためには不要ですが, ヘッダファイルもプロジェクトに追加するため, IDE を使用する場合に便利です。

```
# CMakeLists.txt
cmake_minimum_required (VERSION 2.9)

# プロジェクト名の設定
project (HelloProject)

file(GLOB sourcefiles
        "src/*.hpp"
        "src/*.cpp")

add_executable(Hello ${sourcefiles})
```

■実行時リソース

(DLL, ゲームアセット (game-asset), および, テキストファイルのような) 実行時リソースを実行形式に対する相対パスで読み込むことがあります。

```
CMakeLists.txt
someexe/main.cpp
someexe/res.txt
```

このプロジェクトでは, ソースファイルは, リソースが実行形式と同じディレクトリ内に配置されていることを想定しています。

```
// main.cpp
#include <iostream>
#include <fstream>

int main(){
    std::fstream f("res.txt");
    std::cout << f.rdbuf();
    return 0;
}
```

この CMakeLists.txt はファイルがコピーされることを保証します。

```
# CMakeLists.txt
cmake_minimum_required (VERSION 2.9)

# プロジェクト名の設定
project (HelloProject)

add_executable(Hello someexe/main.cpp)

file(COPY someexe/res.txt DESTINATION Debug)
file(COPY someexe/res.txt DESTINATION Release)
```

注意: このアプローチには, 元のリソースを修正した場合は CMake を再実行しなければならないという問題があります。

■外部ライブラリ

外部ライブラリには, 基本的に 2 つの種類があります。 つまり, バイナリの実行時にリンクされる動的リンクライブラリ (DLL) とコンパイル時にリンクされる静的リンクライブ

ラリです。
静的ライブラリーのセットアップは最も単純です。これを使用するには, コンパイラーがヘッダーファイルの場所を知っている必要があり, リンカーは実際のライブラリの配置位置を知っている必要があります。
外部ライブラリがプロジェクトと一緒に配布されない限り, 通常, それらの場所を知ることはできません - このため, CMake ユーザーが場所を変更できるキャッシュされた変数を使用するのが一般的です。静的ライブラリには, Windows では.lib, その他のほとんどのプラットフォームでは.a で終わるファイルです。

```
# CMakeLists.txt
cmake_minimum_required (VERSION 2.9)

# プロジェクト名の設定
project (HelloProject)


set(fooinclude "/usr/local/include"
        CACHE PATH "ヘッダの配置位置foo")
set(foolib "/usr/local/lib/foo.a"
        CACHE FILEPATH "foo.の配置位置a")

include_directories(${fooinclude})

add_executable(Hello someexe/main.cpp)
target_link_libraries(Hello ${foolib})
```

動的にリンクされたライブラリは, 静的にリンクされたライブラリと同様に機能します。Windows では, コンパイル時にライブラリをリンクする必要がありますが, DLL への実際のリンクは実行時に行われます[*2]。実行可能ファイルは, 実行時リンカーの検索パスから DLL ファイルを検索できる必要があります。DLL がシステムライブラリでない場合, 簡単な解決策は DLL を実行可能ファイルの隣にコピーすることです。多くの場合, DLL の操作にはプラットフォーム固有の操作が必要になります。CMake は, 組み込み変数 WIN32, APPLE, UNIX によりこれらの操作を支援します。

```
# CMakeLists.txt
cmake_minimum_required (VERSION 2.9)

# プロジェクト名の設定
project (HelloProject)

set(fooinclude "/usr/local/include"
        CACHE PATH "ヘッダの配置位置foo")
set(foolib "/usr/local/lib/foo.lib"
        CACHE FILEPATH "foo.の配置位置lib")
set(foodll "/usr/local/lib/foo.dll"
        CACHE FILEPATH "foo.の配置位置dll")

include_directories(${fooinclude})

add_executable(Hello someexe/main.cpp)
target_link_libraries(Hello ${foolib})

IF (WIN32)
    file(COPY ${foodll} DESTINATION Debug)
    file(COPY ${foodll} DESTINATION Release)
ENDIF(WIN32)
```

■ライブラリの自動配置

CMake には, `find_package()` コマンドを使用して, ライブラリを（推奨されるいくつかの場所に基づいて）自動的に検索する機能も含まれています。 ただし, この機能は macOS および Linux で最も適切に機能します。
`https://cmake.org/Wiki/CMake:How_To_Find_Libraries`.

[*2] 訳注: 原文は, “On Windows, it is still needed to link to a library at compile time, but the actual linking to the DLL happens at compile time. ”(「Windows では, コンパイル時にライブラリにリンクする必要がありますが, DLL への実際のリンクはコンパイル時に行われます。」) ですが, “the actual linking to the DLL happens at runtime.”(「DLL への実際のリンクは実行時に行われます。」) の誤記と思われます。

■C++ のバージョン

以下のコマンドで C++ のバージョンを設定することが可能です:

```
set(CMAKE_CXX_STANDARD 14)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

■プリプロセッサシンボルの定義

プロジェクトにプリプロセッサシンボルを追加するには, `add_definitions()` を使用します。

```
# ...
add_definitions(-DFOO=\"XXX\")
add_definitions(-DBAR)
```

これにより, 次のようなソースコード中から使用可能な `FOO` と `BAR` というシンボルが生成されます:

```
#include <iostream>

using namespace std;

int main(){
#ifdef BAR
    cout << "Bar"<< endl;
#endif
    cout << "Hello world "<<FOO << endl;

    return 0;
}
```

■外部リンクと情報

`https://cmake.org/Wiki/CMake/Language_Syntax`
`https://cmake.org/cmake/help/v3.0/command/set.html`

Morten Nobel-Jørgensen(mnob@itu.dk, ITU, 2017) により執筆されました[*3]。

本文書で使用されている LaTeX のテンプレートは, 2015 年, John Smith によるものです。
`http://johnsmith.com/`

[*3] 日本語訳: Takeharu KATO, 2020